A Visual Introduction to OpenGL Programming

Siggraph 1998 - Course 7

Mason Woo
Dave Shreiner

Abstract

This course is an introduction to writing interactive graphics programs using the OpenGL API. As opposed to seeing snippets of code and static, captured images, this course features interactive tools to visualize and experiment with computer graphics concepts, such as transformations, lighting, and texture mapping. Detailed explanations focus upon controlling the position and movement of the camera, the light sources, and objects in a scene. The effects of changing the order of modeling transformations (and their associated matrices) are discussed and visually demonstrated. Manipulating changes to parameters of the texture mapping API are shown with real-time graphics.

Speaker Biographies

Mason Woo

After 10 years of training and marketing graphics libraries at Silicon Graphics, Mason Woo became an independent consultant in 1996. He is co-author of the OpenGL Programming Guide (Addison-Wesley, 2nd edition, 1997) and former secretary of the OpenGL Architecture Review Board. Mason has previously taught courses at SIGGRAPH, the X Technical Conference, and Xhibition, and has been a speaker or panelist at JavaOne, the Japan Personal Computer Software Association, NCGA, VESA, Microsoft Win32 Professional Developer's Conference, Defense & Government Computer Graphics Conference, and SIGCHI & GI.

Dave Shreiner

Dave is a member of the OpenGL development team at Silicon Graphics Computer Systems. He has 10 years of experience with visual simulation and scientific visualization, including 7 years at Silicon Graphics. He was the original author of “Introductory OpenGL Programming” for Silicon Graphics Technical Education department. Dave has a Bachelors of Mathematics from University of Delaware, and has done graduate work at the Johns Hopkins University.

Syllabus

1:30pm	Woo	Welcome & OpenGL Introduction (pages 1-14)
1:50pm	Shreiner	Elementary Rendering (pages 15-36)
2:20pm	Woo	Matrix Transformations (pages 37-56)
3:00pm	Break	
3:15pm	Shreiner	Lighting Models (pages 57-68)
3:45pm	Shreiner	Texturing (pages 69-84)
4:25pm	Woo	Overview of Other Topics (pages 85-106)
5:15pm	All	Summay, Q & A (pages 107-109)

Table of Contents

A Visual Introduction to OpenGL Programming

Mason Woo
Dave Shreiner

1

Course Flow

Introduction

- What is OpenGL?
- OpenGL Architecture
- OpenGL and the window system
- Typical program structure

Elementary Rendering

- Geometry Primitives
- States
- Animation
- Buffers

2

Course Flow

Viewing and Transformations

Lighting

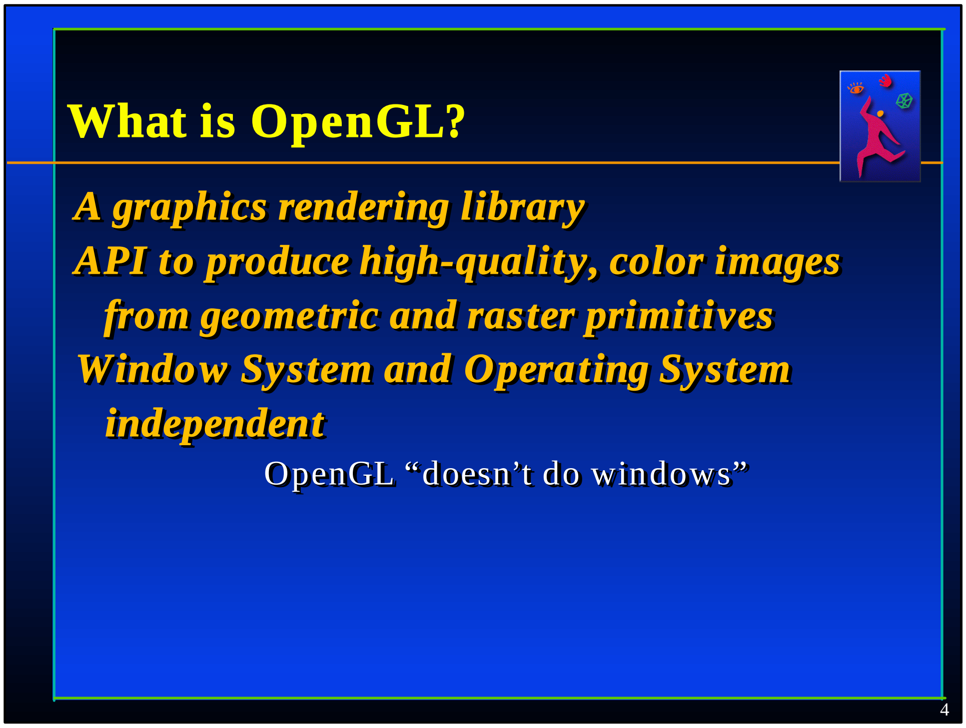
Texture mapping

Overview of Other Operations

- Display Lists
- Feedback
- Image Primitives
- Fog, Antialiasing, Fragment operations

Summary and References, Q&A

3



A graphics rendering library is a layer of abstraction between graphics hardware and an application program.

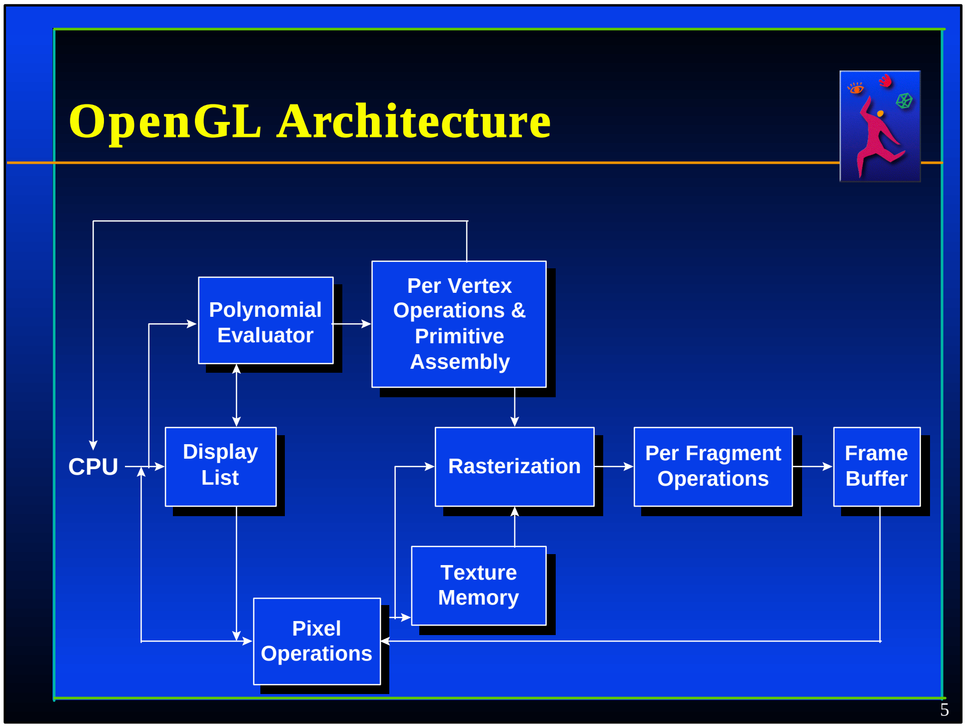
API = Application Programming (or Procedural) Interface.

Geometric primitives are vertex-based and are either 2D or 3D.

Raster primitives are pixel-based (either bitmaps or pixmaps) and generally 2D. Texture mapping combines both raster and geometric primitives to create an image.

OpenGL libraries are supported for use with X Window System® and UNIX®, Microsoft Windows®, Microsoft Windows NT®, and IBM OS/2® .

OpenGL does not perform operations which are redundant with the window system: window management, event (mouse & keyboard) handling, and loading color maps.



This is the most important diagram you will see, representing the flow of graphical information, as it is processed from CPU to the frame buffer.

There are two pipelines of data flow. The upper pipeline is for geometric, vertex-based primitives. The lower pipeline is for pixel-based, image primitives. Texturing combines the two types of graphics together.

There is a pull-out poster in the back of the OpenGL Reference Manual (blue book), which shows this diagram in more detail.

OpenGL as a Renderer

Renders simple geometric primitives

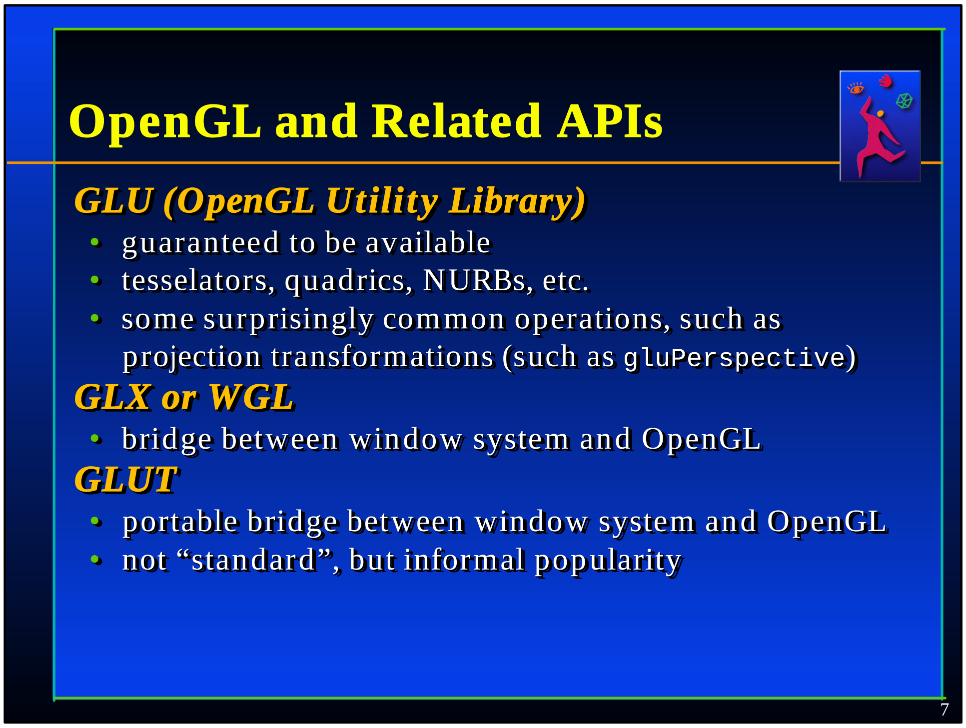
- points, lines, polygons

Renders images and bitmaps

- separate pipelines for geometry and pixels, linked through texture mapping

Rendering depends on state

- colors, light sources, materials
- surface normals, texture coordinates

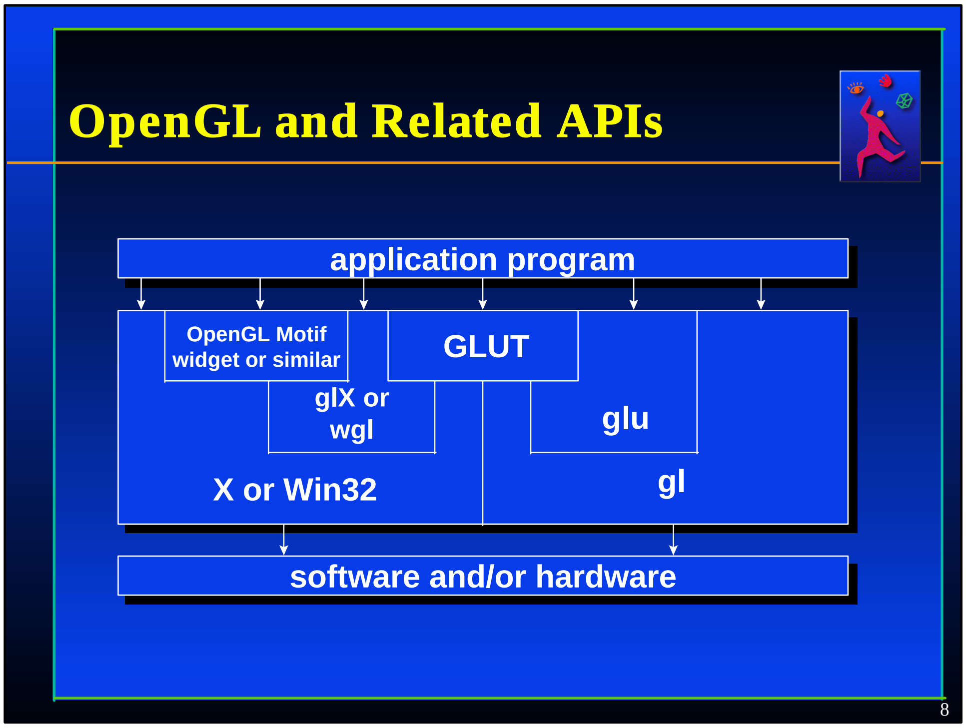


The GLU (OpenGL Utility library) is a set of commonly used graphics routines mandatory for *all* implementations of OpenGL. All routines in the GLU have the prefix, *glu*. The GLU contains more complicated commands, such as tesselators, quadric objects, and NURBS.

The GLX and WGL libraries are extensions of the X Window System and Microsoft Windows, respectively. They support operations to create an OpenGL context, visuals (or corresponding pixel format), frame buffer configuration (including double buffering and depth buffer size), and synchronization.

The GLUT (OpenGL Utility Toolkit) is a set of portable, convenience routines to deal with window management, event handling, and modeling some basic 3D objects. Implementations of GLUT have been ported to different window systems, including both X and Microsoft Windows, so programs written with GLUT port very easily. GLUT is not an official, governed API; it was originally written by Mark Kilgard and has gained informal acceptance in the OpenGL community.

Mark Kilgard’s book, *OpenGL Programming for the X Window System*, is published by Addison-Wesley (ISBN 0-201-48359-9). His book includes extensive description of the GLUT toolkit.


OpenGL and Related APIs
application program
OpenGL Motif
widget or similar
GLUT
glX or
wgl
glu
X or Win32
gl
software and/or hardware
8

Program Structure

initialize visual & open window
initialize states & display lists
register display callback function

- clear screen, change states, render graphics, swap buffers...

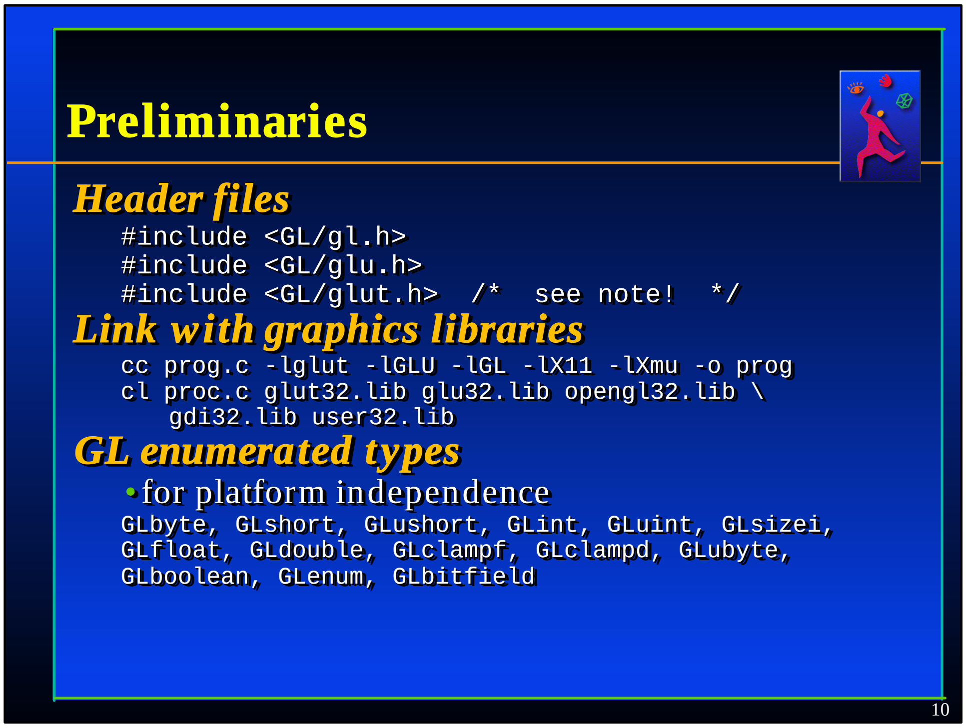
register reshape callback function

- modify clipping, viewing

register input device (mouse, keybd) callback functions
register idle callback function (the *keep busy* operation)
enter main loop

- if contents need to be redrawn, display callback called
- if window resized, reshape callback called
- if input event, appropriate input callback function called
- if nothing happening, idle callback function called

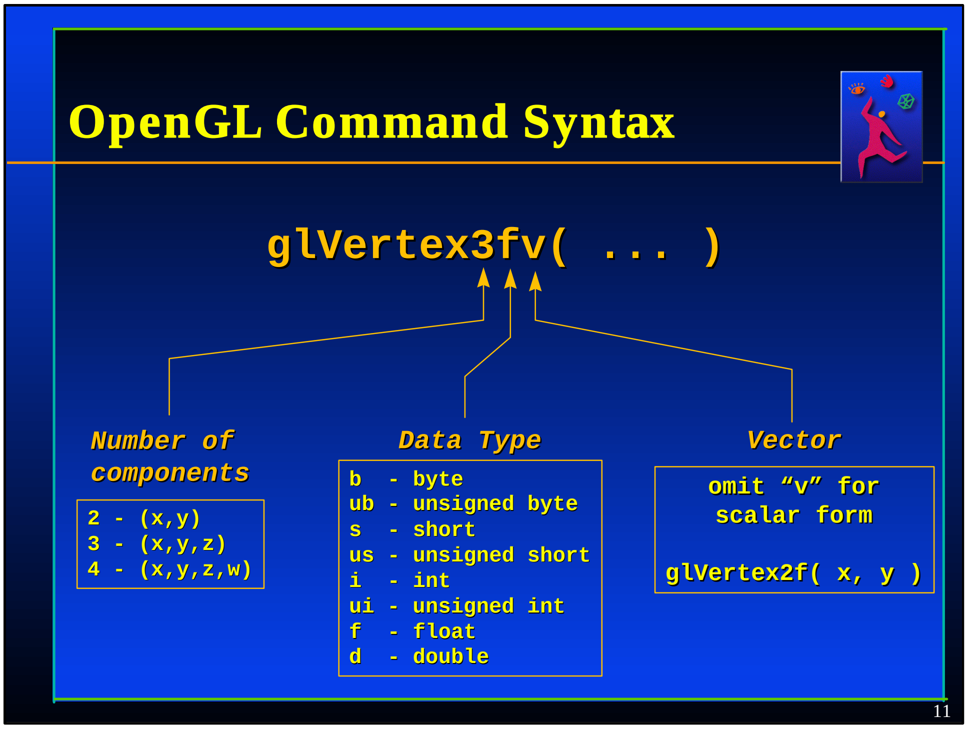
9



Note that including glut.h automatically includes both gl.h and glu.h. Also, for Microsoft Windows, glut.h includes windows.h along with gl.h and glu.h, so that no compiler errors or warnings result. Therefore, if you use glut.h, it is recommended that you *don't* redundantly include gl.h and glu.h again.

For Microsoft Windows, be sure the INCLUDE and LIB environment variables are pointing to correct path. One common setting is:

```
set INCLUDE=c:\msdev\include
set LIB=c:\msdev\lib
```



The OpenGL API calls are designed to accept almost any basic data type, which is reflected in the call's name. Knowing how the calls are structured makes it easy to determine which call should be used for a particular data format and size.

For instance, vertices from most commercial models are stored as three component, single-precision, floating point vectors. As such, the appropriate OpenGL command to use is `glVertex3fv( coords )`.

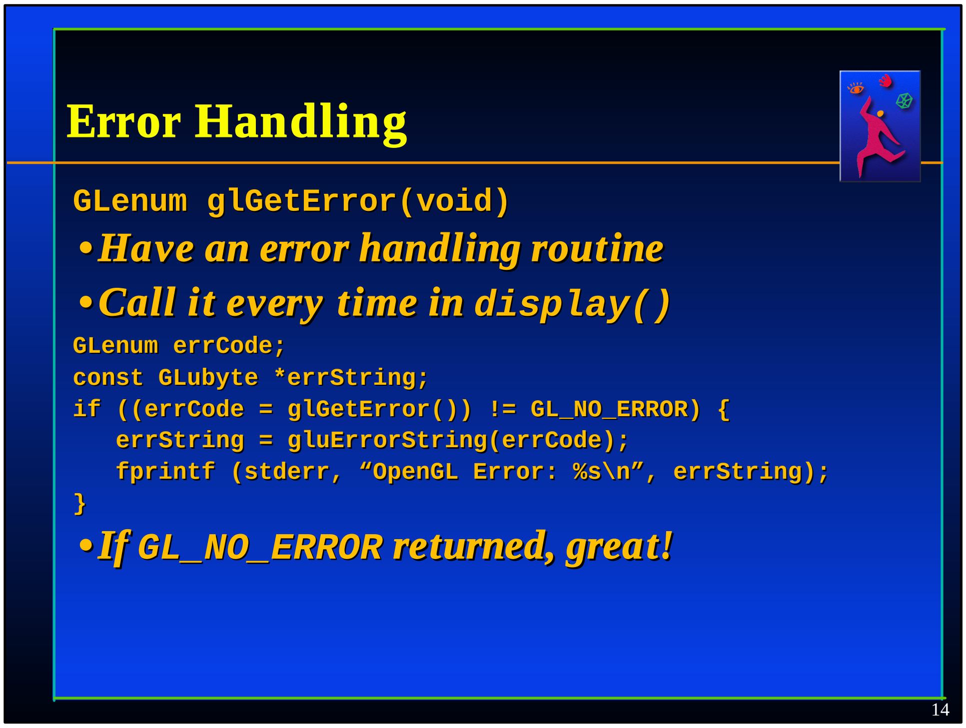
OpenGL uses homogenous coordinates to specify vertices. For `glVertex*()` calls which don't specify all the coordinates (i.e. `glVertex2f()`), OpenGL defaults to z = 0.0, and w = 1.0

A Simple Example Program

```
#include <GL/glut.h>
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(0.0, 1.0, 1.0);   /* cyan */
  glBegin(GL_QUADS);
  glVertex2i(100,100);
  glVertex2i(200, 100);
  glVertex2i(200,300);
  glVertex2i(100, 300);
  glEnd();
  glFlush();
}
void gfxinit(void) {
  glClearColor(0.0, 0.0, 0.0, 0.0);
}
```

Simple Program (continued)

```
void reshape(int width, int height) {
  glViewport(0, 0, width, height)
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0.0, (GLdouble) width, 0.0,
      (GLdouble) height);
}
void main(int argc, char **argv){
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  win = glutCreateWindow("rect");
  glutReshapeFunc(reshape);
  glutDisplayFunc(display);
  gfxinit();
  glutMainLoop();
}
```

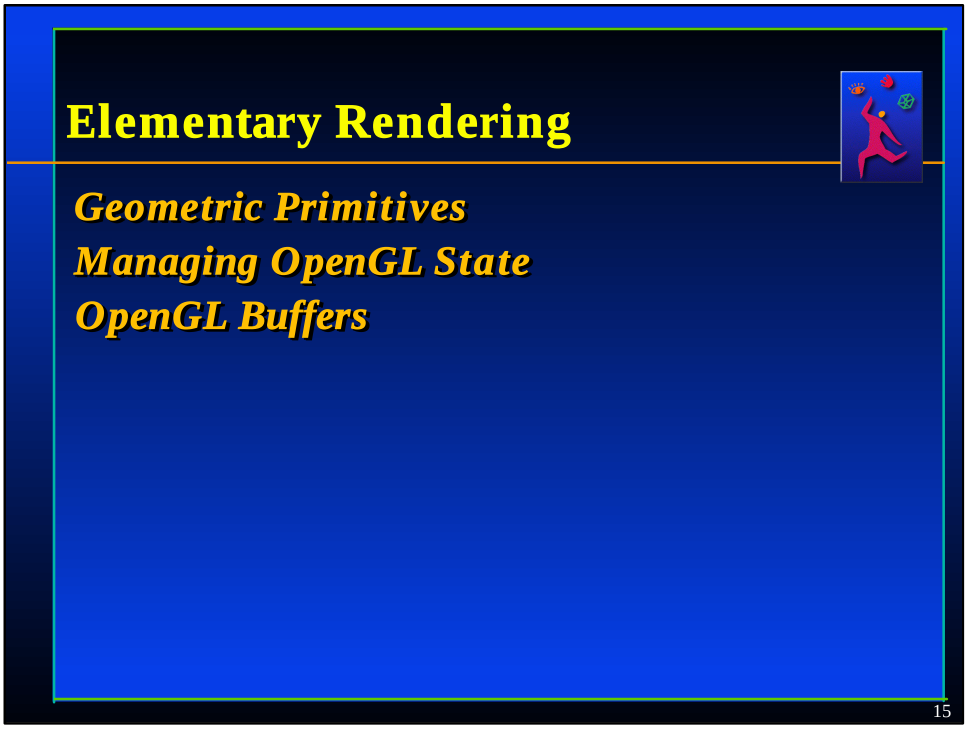


`glGetError()` can also return: `GL_STACK_OVERFLOW`, `GL_STACK_UNDERFLOW`, `GL_INVALID_VALUE`, `GL_INVALID_OPERATION`, `GL_INVALID_ENUM`, or `GL_OUT_OF_MEMORY`.

`gluErrorString()` converts the returned error into something printable.

There is also `gluGetError()` for error conditions in the GLU.

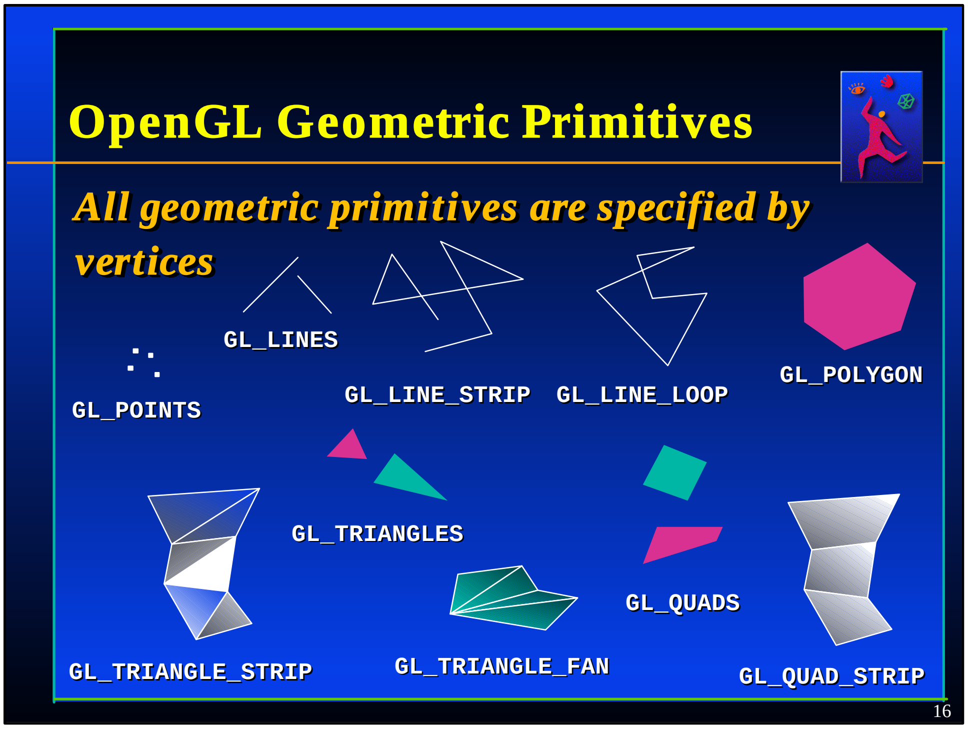
For a typical, single-threaded OpenGL implementation, only one error is recorded. If multiple errors occur, only the first error is recorded. For distributed implementations (such as multi-threaded), there may be several errors recorded.



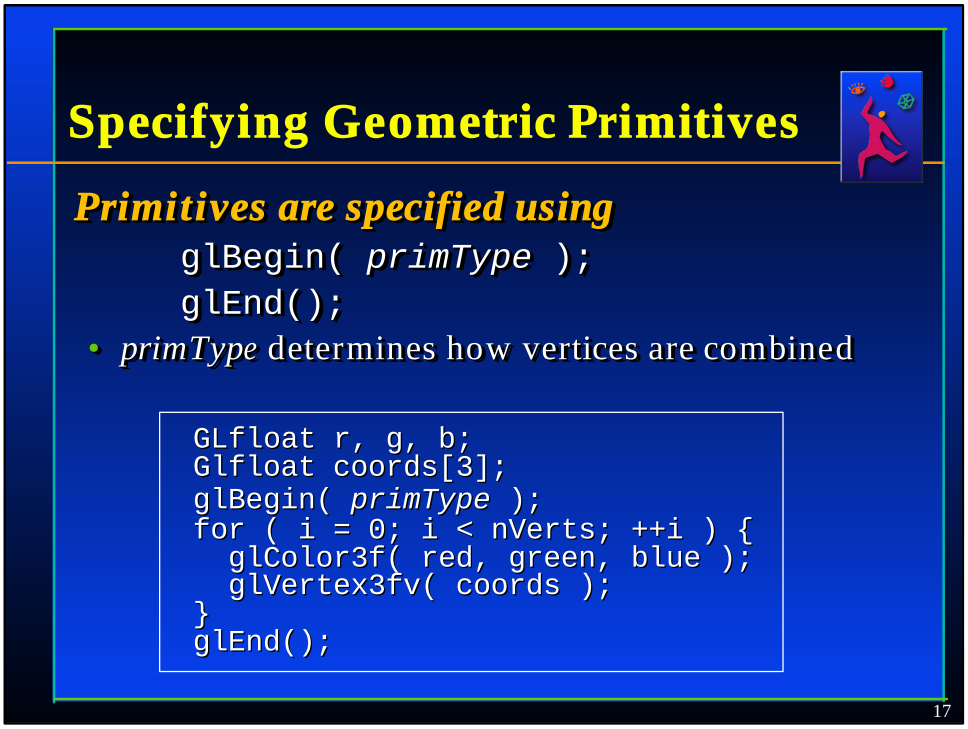
In this section, we'll be discussing the basic geometric primitives that OpenGL uses for rendering, as well as how to manage the OpenGL state which controls the appearance of those primitives.

OpenGL also supports the rendering of bitmaps and images, which is discussed in a later section.

Additionally, we'll discuss the different types of OpenGL buffers, and what each can be used for.

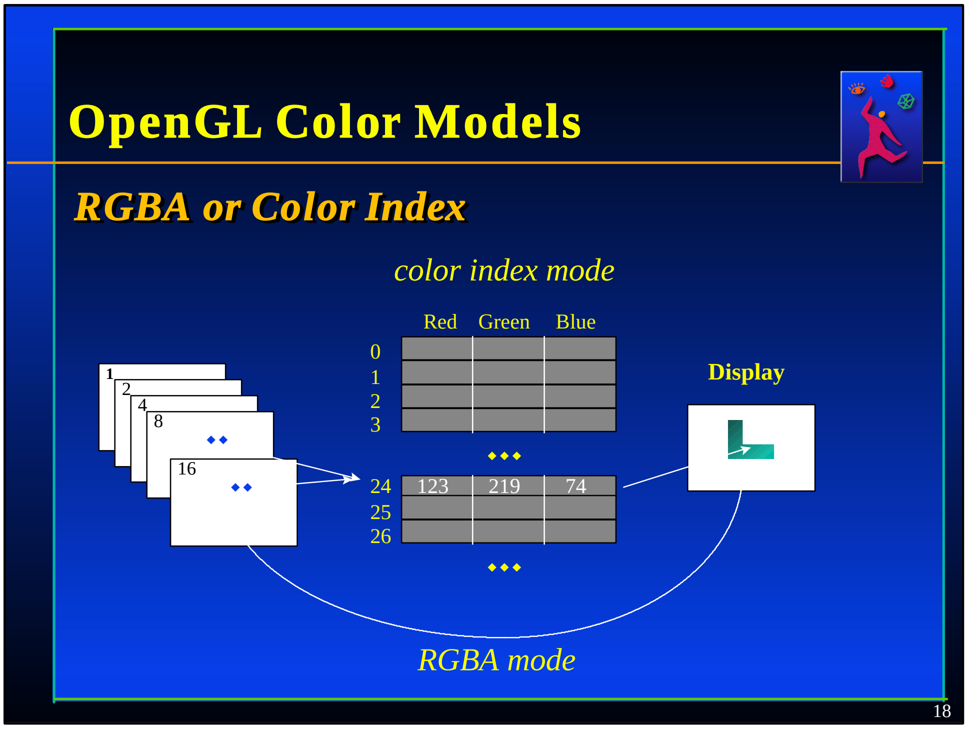


Every OpenGL geometric primitive is specified by its vertices, which are *homogenous coordinates*. Homogenous coordinates are of the form (x, y, z, w). Depending on how vertices are organized, OpenGL can render any of the shown primitives.



OpenGL organizes vertices into primitives based upon which type is passed into `glBegin()`. The possible types are:

```
GL_POINTS          GL_LINE_STRIP
GL_LINES           GL_LINE_LOOP
GL_POLYGON
GL_TRIANGLE_STRIP
GL_TRIANGLES       GL_TRIANGLE_FAN
GL_QUADS           GL_QUAD_STRIP
```



Every OpenGL implementation must support rendering in both RGBA mode, (sometimes described as *TrueColor* mode) and color index (or *colormap*) mode.

For RGBA rendering, vertex colors are specified using the `glColor*()` call.

For color index rendering, the vertex's index is specified with `glIndex*()`.

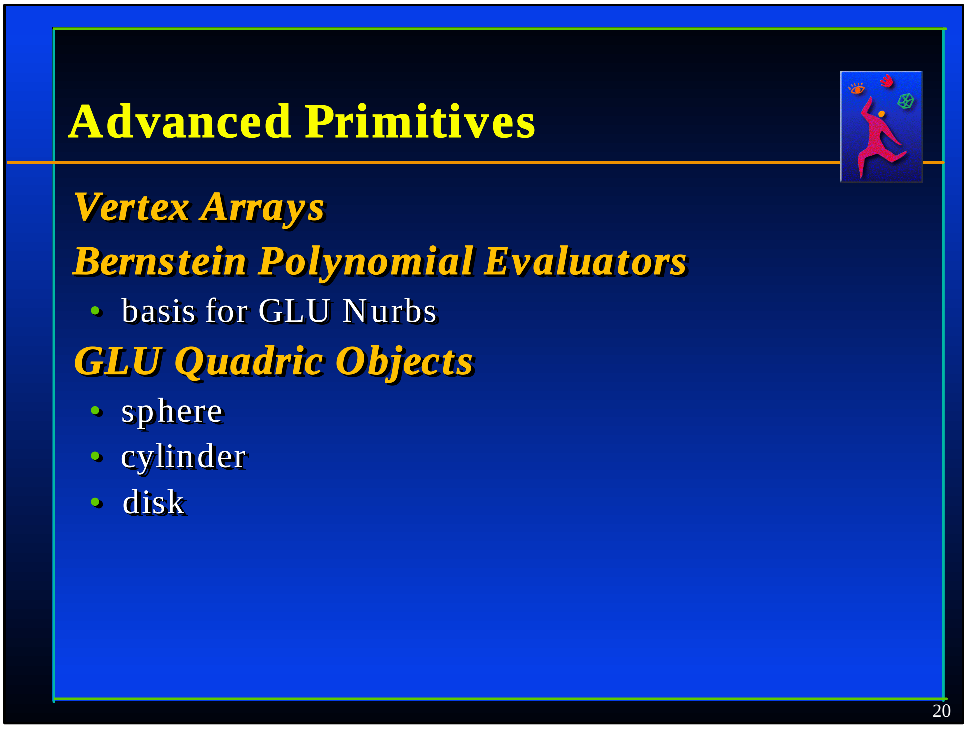
The type of window color model is requested from the windowing system. Using GLUT, the `glutInitDisplayMode()` call is used to specify either an RGBA window (using `GLUT_RGBA`), or a color indexed window (using `GLUT_INDEX`)

Simple Example

```
void drawRhombus( GLfloat color[] )
{
  glColor3fv( color );
  glBegin( GL_QUADS );
  glVertex2f( 0.0, 0.0 );
  glVertex2f( 1.0, 0.0 );
  glVertex2f( 1.5, 1.118 );
  glVertex2f( 0.5, 1.118 );
  glEnd();
}
```

19

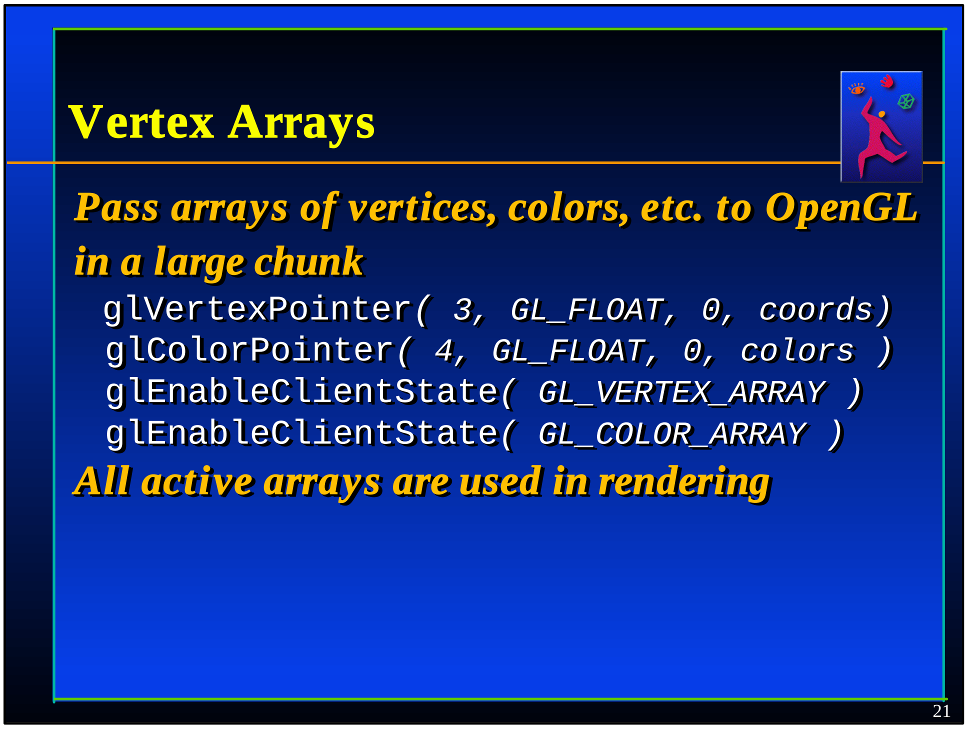
The `drawRhombus()` routine causes OpenGL to render a single quadrilateral in a single color. The rhombus is planar, since the *z* value is automatically set to 1.0 by `glVertex2f()`.



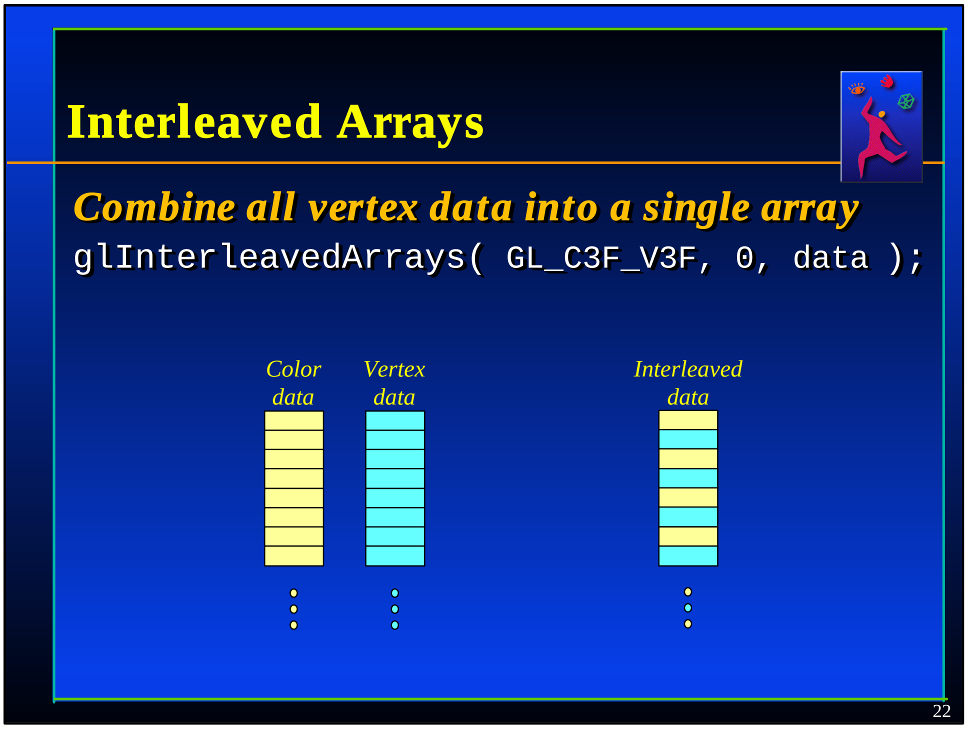
In addition to specifying vertices one at a time using `glVertex*()`, OpenGL supports the use of arrays, which allows you to pass an array of vertices, lighting normals, colors, edge flags, or texture coordinates. This is very useful for systems where functions calls are computationally expensive. Additionally, the OpenGL implementation may be able to optimize the processing of arrays.

OpenGL evaluators, which automate the evaluation of the Bernstein polynomials, allow curves and surfaces to be expressed algebraically. They are the underlying implementation of the OpenGL Utility Library's NURBS implementation.

Finally, the OpenGL Utility Library also has calls for generating polygonal representation of quadric objects. The calls can also generate lighting normals and texture coordinates for the quadric objects.



Vertex Arrays allow vertices, and their attributes to be specified in chunks, which reduces the need for sending singule vertices and their attributes one call at a time. This is a useful optimization technique, as well as usually simplifying storage of polygonal models.



`glInterleavedArrays()` combines all the information for a vertex together locally in memory. When the array is traversed, the *format* (`GL_C3F_V3F` in the above example) tells OpenGL what data, and how it should be processed, are stored in the array.

The supported formats for `glInterleavedArrays()` are:

`GL_V2F`	`GL_C4UB_V2F`	`GL_C3F_V3F`
`GL_V3F`	`GL_C4UB_V3F`	`GL_N3F_V3F`
`GL_C4F_N3F_V3F`	`GL_T2F_C4UB_V3F`	`GL_T2F_V3F`
`GL_T2F_C3F_V3F`	`GL_T2F_N3F_V3F`	`GL_T4F_V4F`

`GL_T2F_C4F_N3F_V3F`   `GL_T4F_C4F_N3F_V4F`

where the letters below represent the following vertex data:

V - Vertex coordinates

C - Color information

N - Normal vectors

T - Texture coordinates

Rendering with Vertex Arrays

Render arrays sequentially

```
glDrawArrays( GL_TRIANGLE_STRIP, 0, numVerts );
```

Automatically index into arrays

```
Gluint indices[] = { 0, 2, 1, 3, 2, 3, 6, 5 };
glDrawElements( GL_QUADS, 2, GL_UNSIGNED_INT,
  indices );
```

Manually index into arrays

```
glBegin( GL_LINES );
for ( i = 0; i < numLines; ++i ) {
   glArrayElement( leftEnd[i] );
   glArrayElement( rightEnd[i] );
}
glEnd();
```

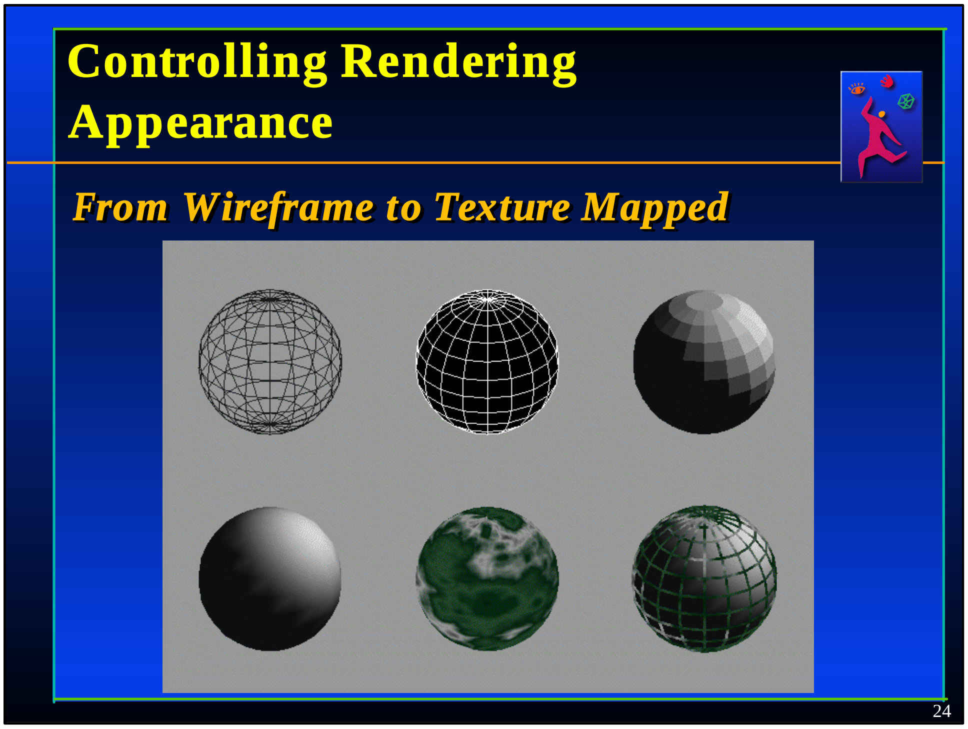
23

When OpenGL processes the arrays, any enabled array is used for rendering. There are three methods for rendering using vertex arrays:

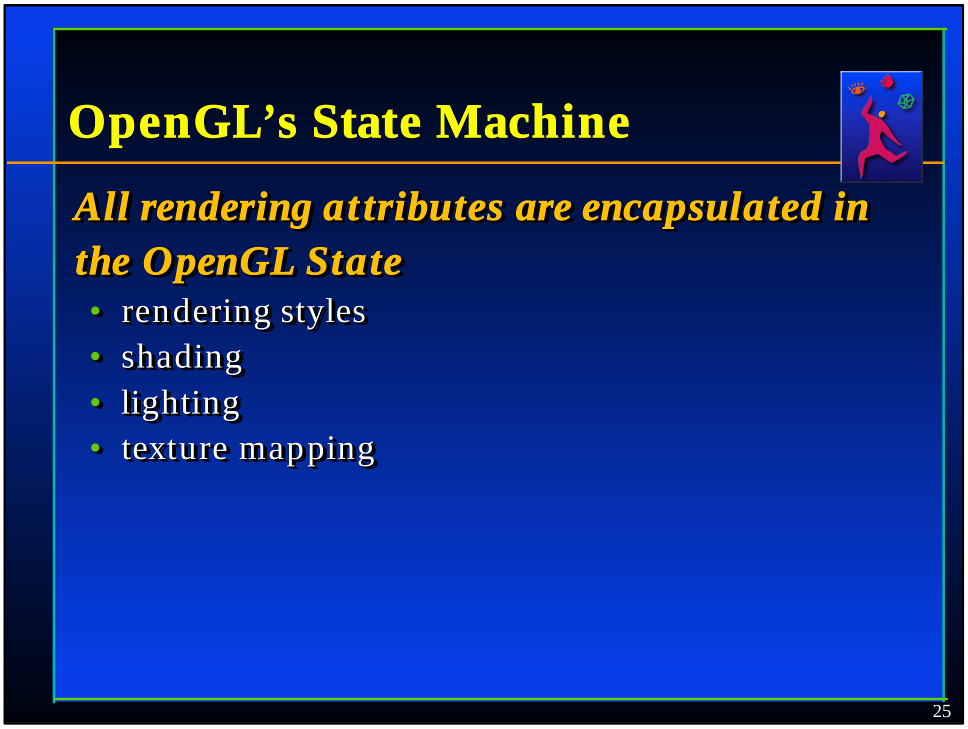
First, the `glDrawArrays()` routine will render the specified primitive type by processing *numVerts* consecutive data elements from the enabled arrays.

Second, `glDrawElements()` allows indirect indexing of data elements in the enabled arrays. This allows shared data elements to be specified only once in the arrays, but be accessed numerous times.

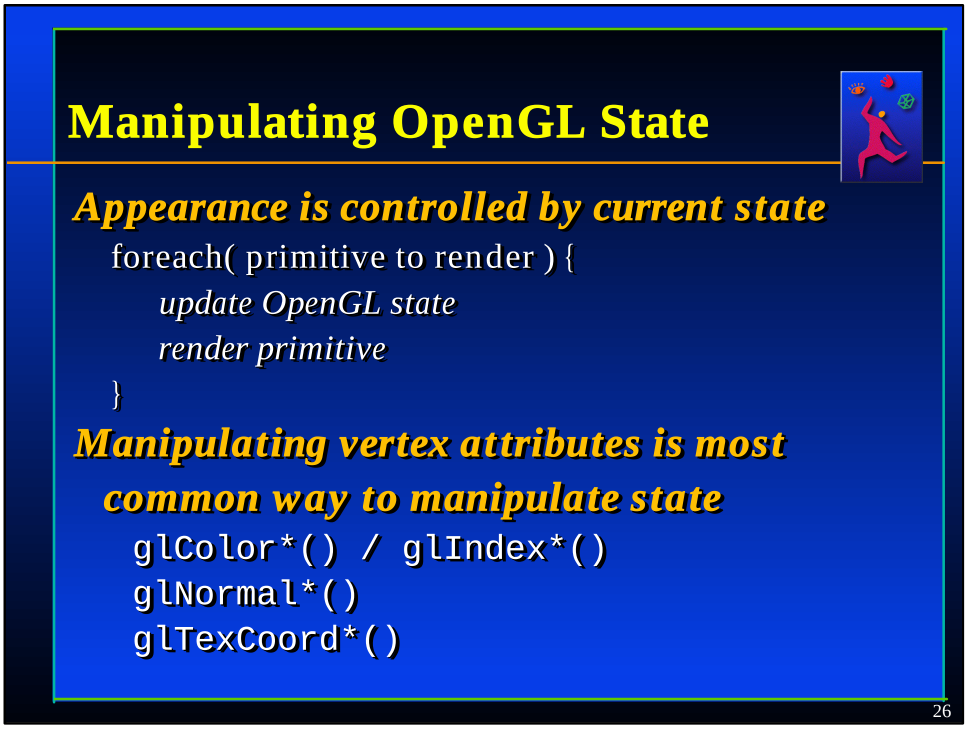
Finally, `glArrayElement()` processes a single set of data elements from all activated arrays. As compared to the previous two commands above, `glArrayElement()` must appear between a `glBegin()` / `glEnd()` pair.



OpenGL can render from a simple line-based wireframe to complex multi-pass texturing algorithms to simulate bump mapping or Phong lighting.

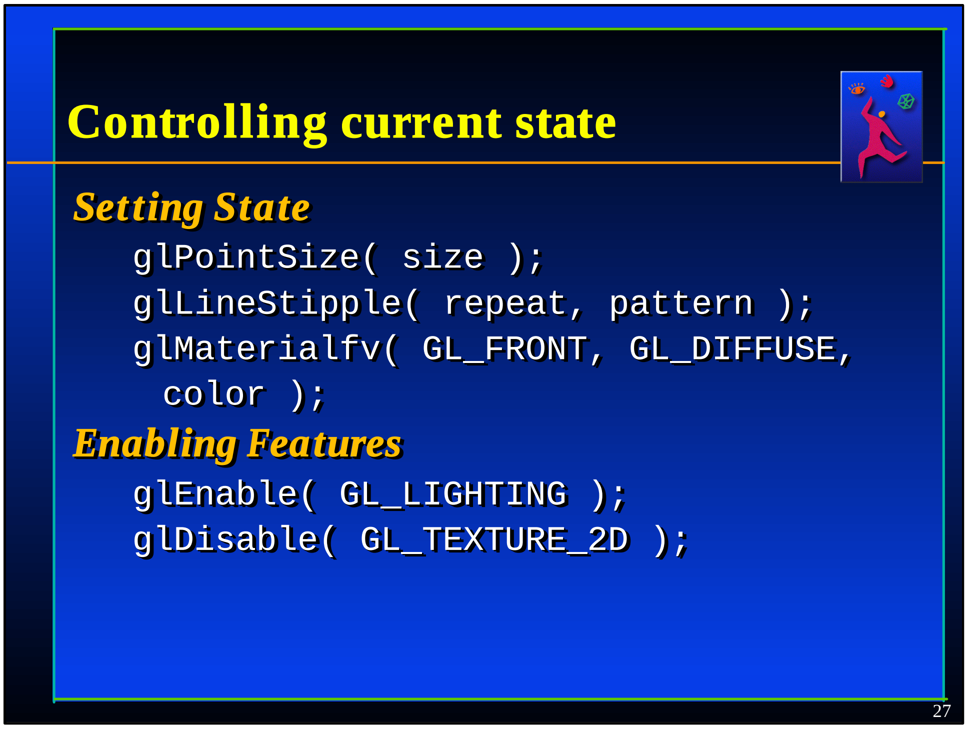


Each time OpenGL processes a vertex, it uses data stored in its internal state tables to determine how the vertex should be transformed, lit, textured or any of OpenGL's other modes.



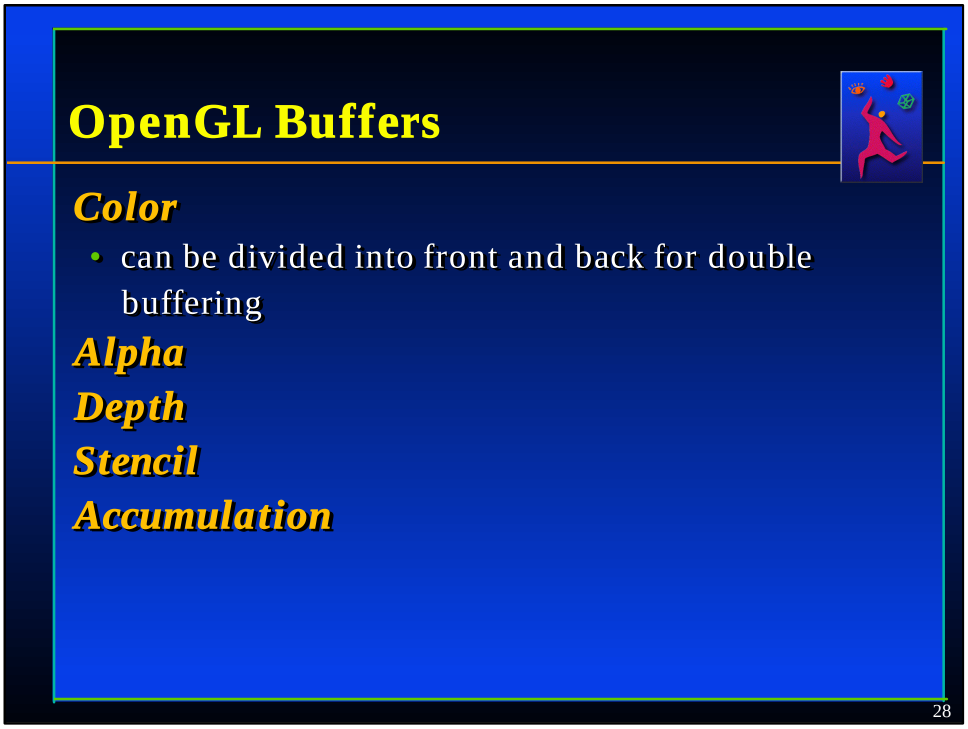
The general flow of any OpenGL rendering is to set up the required state, then pass the primitive to be rendered, and repeat for the next primitive.

In general, the most common way to manipulate OpenGL state is by setting vertex attributes, which include color, lighting normals, and texturing coordinates.



Setting OpenGL state usually includes modifying the rendering attribute, such as loading a texture map, or setting the line width. Also for some state changes, setting the OpenGL state also enables that feature (like setting the point size or line width).

Other features need to be turned on. This is done using `glEnable()`, and passing the token for the feature, like `GL_LIGHT0` or `GL_POLYGON_STIPPLE`.



OpenGL supports a variety of different buffers, some of which hold color data, and others which control the updating of that color data.

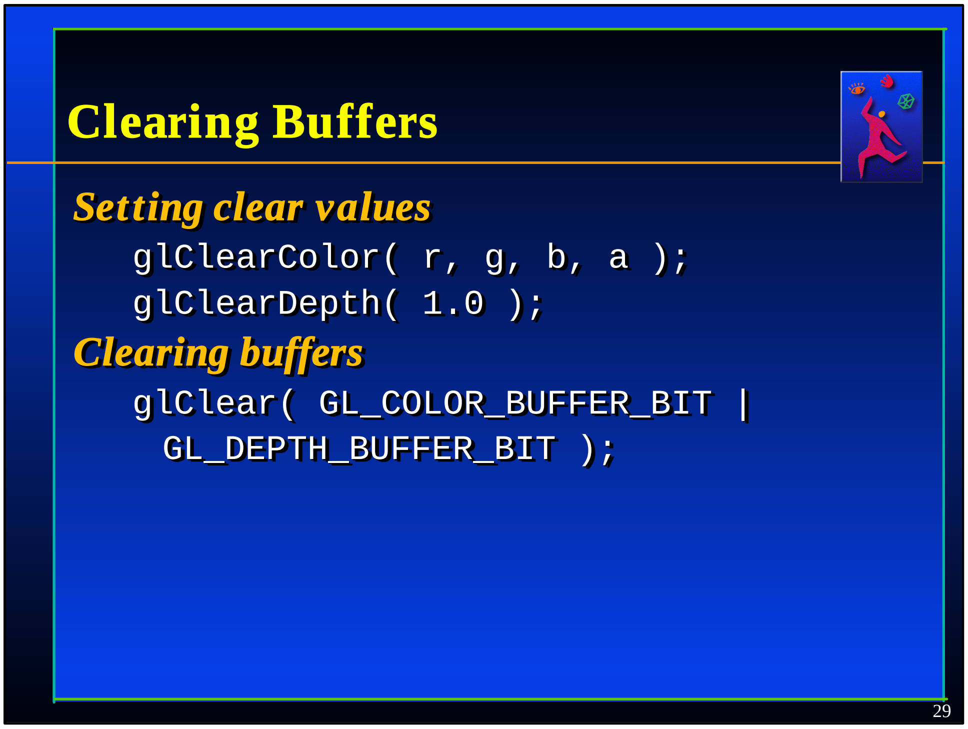
The principle OpenGL buffer is the *color buffer*, which is generally the hardware framebuffer, or an offscreen rendering pixmap.

The *depth buffer*, which is also called the *z*-buffer, is used for visible surface determination.

The *stencil buffer* provides a per-pixel mask test to determine if a pixel in the color buffer should be updated.

Finally, the *accumulation buffer* is used for the composting of several renderings from the color buffer, which can be scaled and combined together to produce a final image which is transferred back to the color buffer for display.

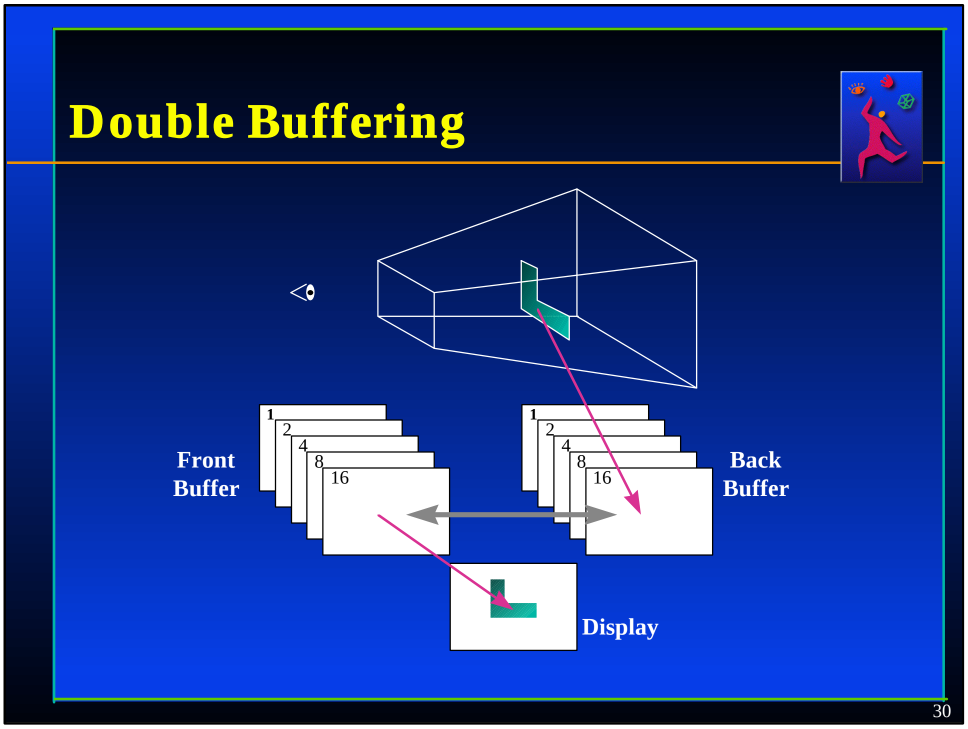
The stencil and accumulation buffers are somewhat advanced topics, and are outside the scope of this course.



The `glClear*()` commands are used to set the default values used to clear each of a windows active buffers to. When the `glClear()` call is executed, it clears each of the buffers requested with their clear values.

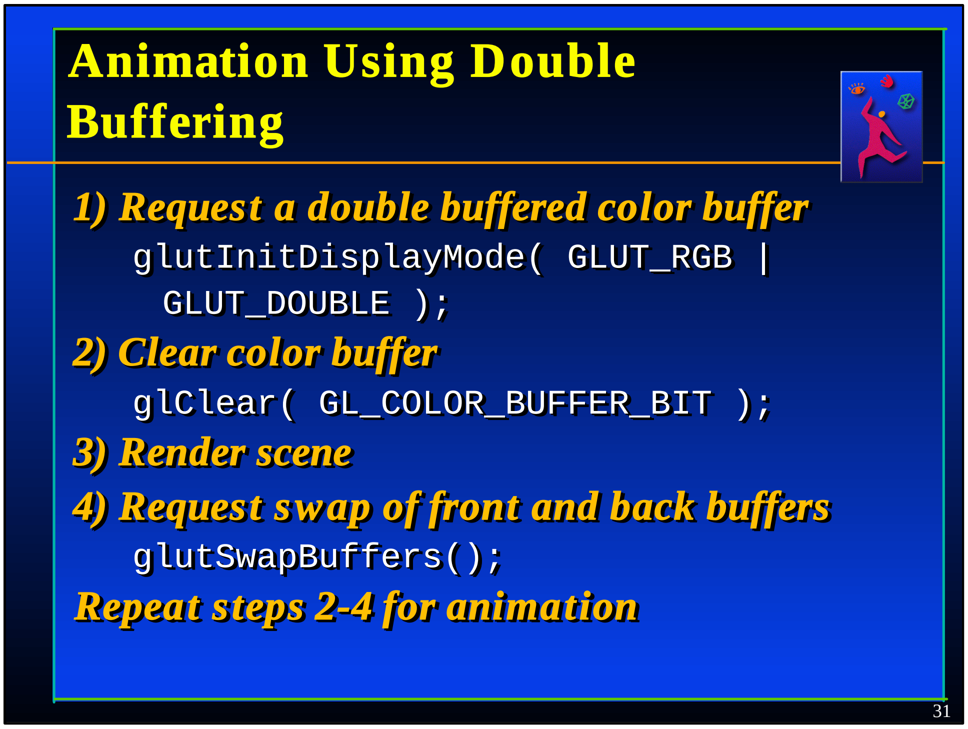
The commands for setting the default clear values are:

- `glClearColor()` - set default RGBA value for color buffer (RGBA mode)
- `glClearIndex()` - set default color index for color buffer (Color index mode)
- `glClearDepth()` - set default depth value for depth buffer
- `glClearAccum()` - set default color for accumulation buffer
- `glClearStencil()` - set default value for stencil buffer



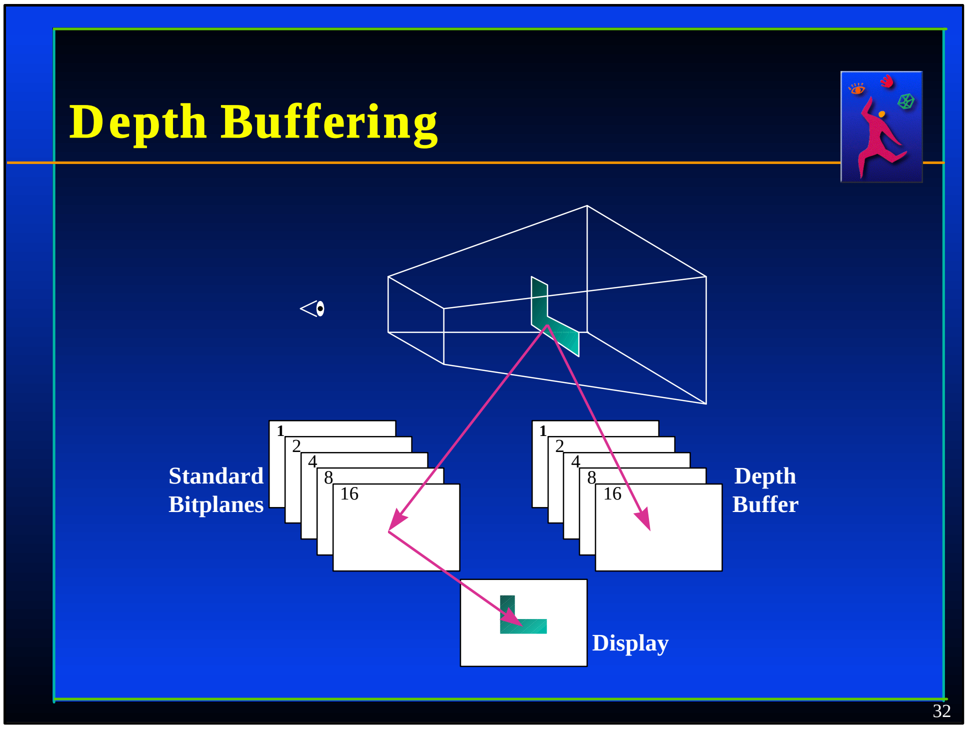
Double buffer is a technique for tricking the eye into seeing smooth animation of rendered scenes. The color buffer is usually divided into two equal halves, called the *front buffer* and the *back buffer*.

The front buffer is displayed while the application renders into the back buffer. When the application completes rendering to the back buffer, it requests the graphics display hardware to swap the roles of the buffers, causing the back buffer to now be displayed, and the previous front buffer to become the new back buffer.



Requesting double buffering in GLUT is simple. In addition to specifying what type of color model you would like to use, adding `GLUT_DOUBLE` to your `glutInitDisplayMode()` call will cause your window to be double buffered.

When your application is finished rendering its current frame, and wants to swap the front and back buffers, the `glutSwapBuffers()` call will request the windowing system to update the window's color buffers.

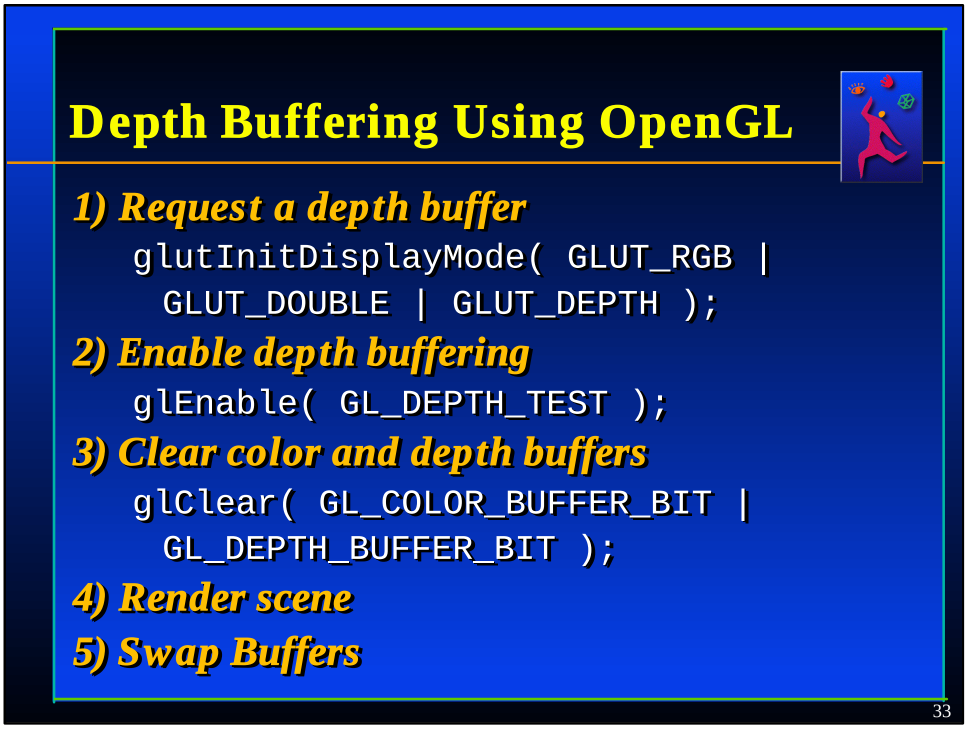


Depth buffering is a technique for determine which primitives in your scene are occluded by other primitives. As each pixel in a primitive is rasterized, its distance from the eyepoint (depth value), is compared with the values stored in the depth buffer. If the pixel's depth value is less than the stored value, the pixel's depth value is written to the depth buffer, and its color is written to the color buffer.

The depth buffer algorithm is generally:

```
if ( pixel->z < depthBuffer(x,y)->z ) {
     depthBuffer(x,y)->z = pixel->z;
     colorBuffer(x,y)->color = pixel->color;
 }
```

OpenGL depth values range from [0, 1], with one being essientially infinitely from the eyepoint. Generally, the depth buffer is cleared to one at the start of a frame.



Enabling depth testing in OpenGL is very straightforward.

A depth buffer must be requested for your window, once again using the `glutInitDispayMode()`, and the `GLUT_DEPTH` bit.

Once the window is created, the depth test is enabled using `glEnable( GL_DEPTH_TEST )`.

A Complete Example

```
void main( int argc, char** argv )
{
   glutInit( &argc, argv );
   glutInitDisplayMode( GLUT_RGB |
      GLUT_DOUBLE | GLUT_DEPTH );
   glutCreateWindow( “Tetrahedron” );
   init();
   glutIdleFunc( drawScene );
   glutMainLoop();
}
```

34

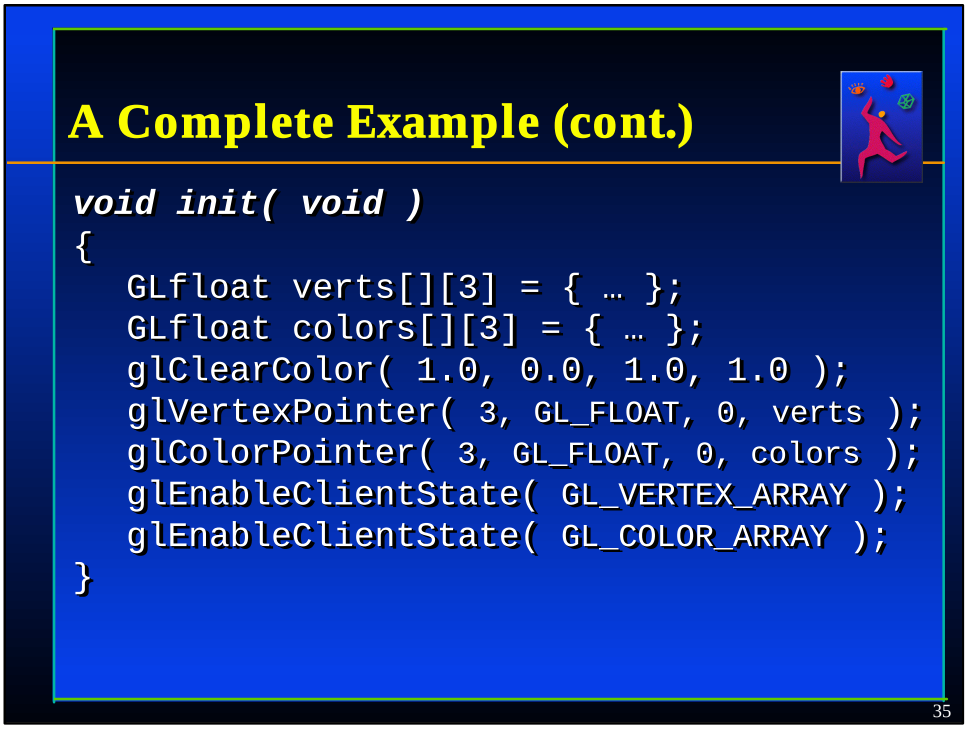
In `main()`,

1) GLUT initializes and creates a window named “Tetrahedron”

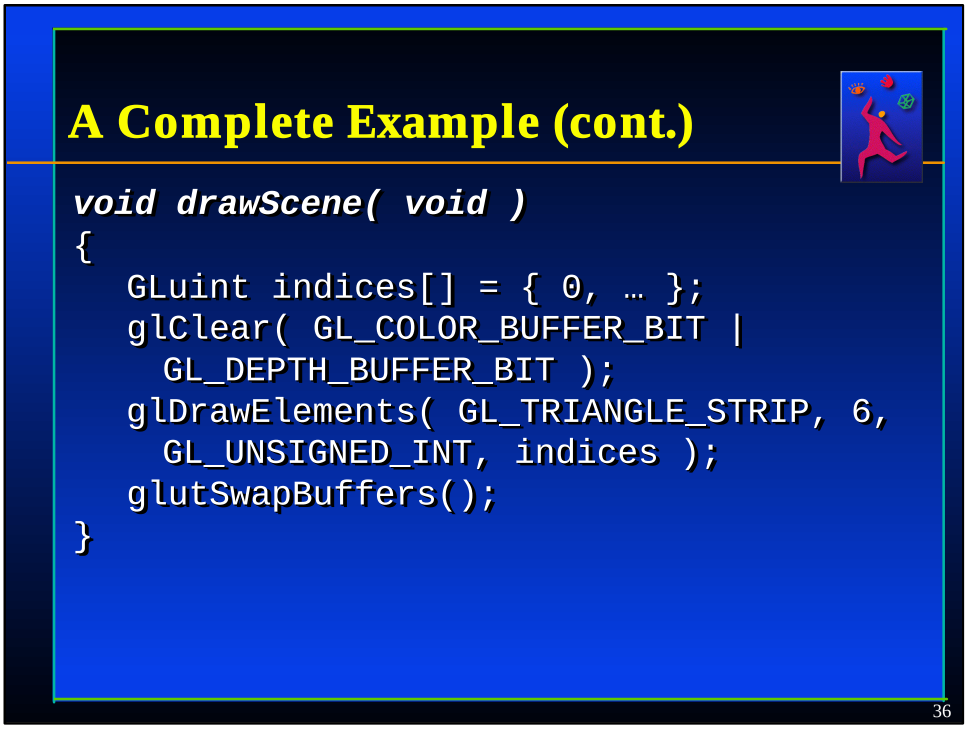
2) set OpenGL state which is enabled through the entire life of the program in
   `init()`

3) set GLUT’s idle function, which is used executed when there are no user user events to process

4) enter the main event processing loop of the program.



In `init()` the basic OpenGL state to be used throughout the program is initialized. Since vertex arrays are used, they are set and enabled in `init()`, and referenced when rendering occurs. Additionally, we set the background (clear color) for the color buffer.



In `drawScene()`,

1) the color buffer is cleared to the background color
2) a triangle strip is rendered to create a tetrahedron using `glDrawElements()` and the vertex arrays that were set up in `init()`.
3) the front and back buffers are swapped.

Transformations

Prior to rendering, view, locate, and orient:

- eye/camera position
- 3D geometry

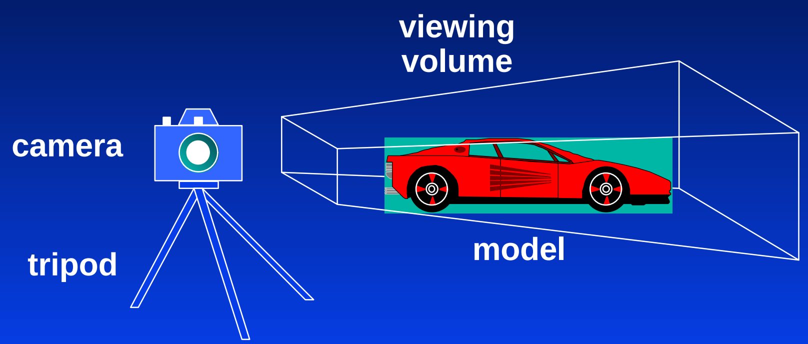
Manage the matrices

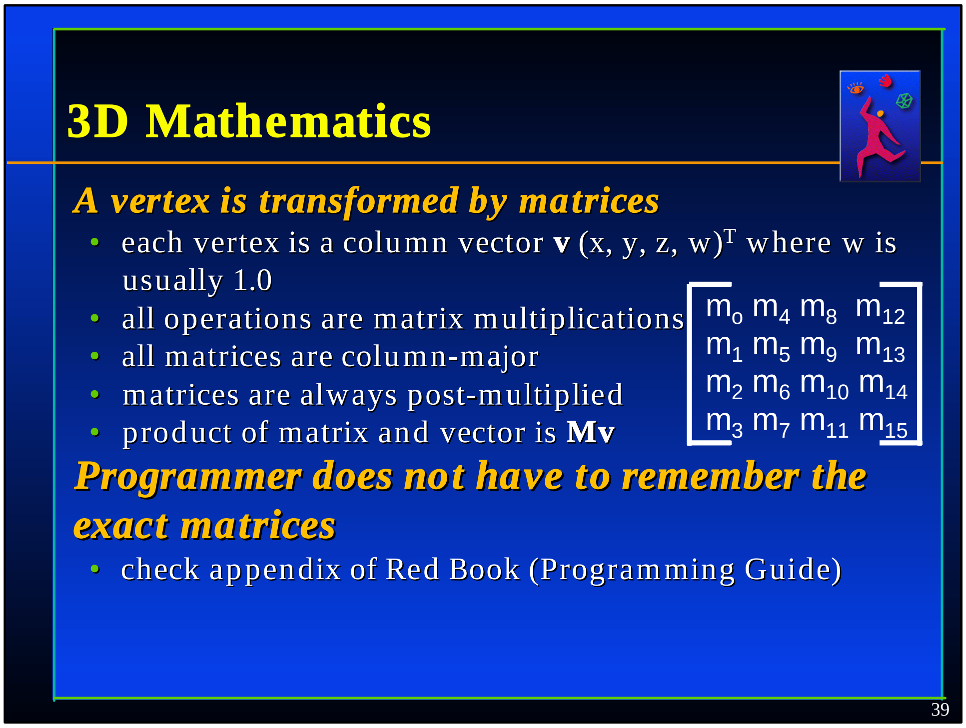
- including matrix stack

Combine (composite) transformations

Camera Analogy

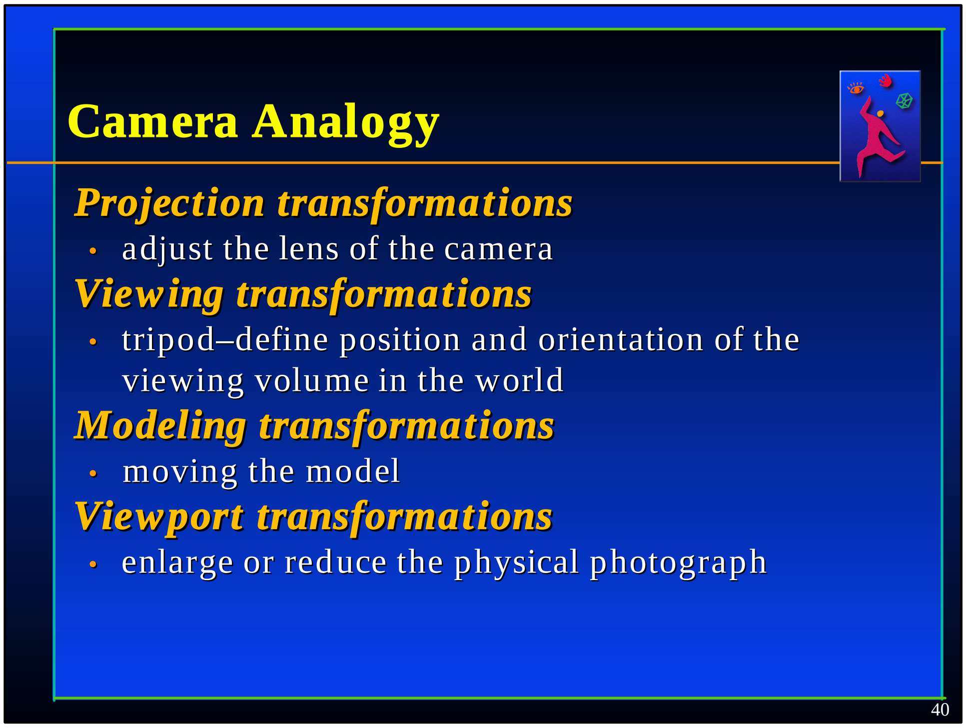
3D is just like taking a photograph (lots of photographs!)



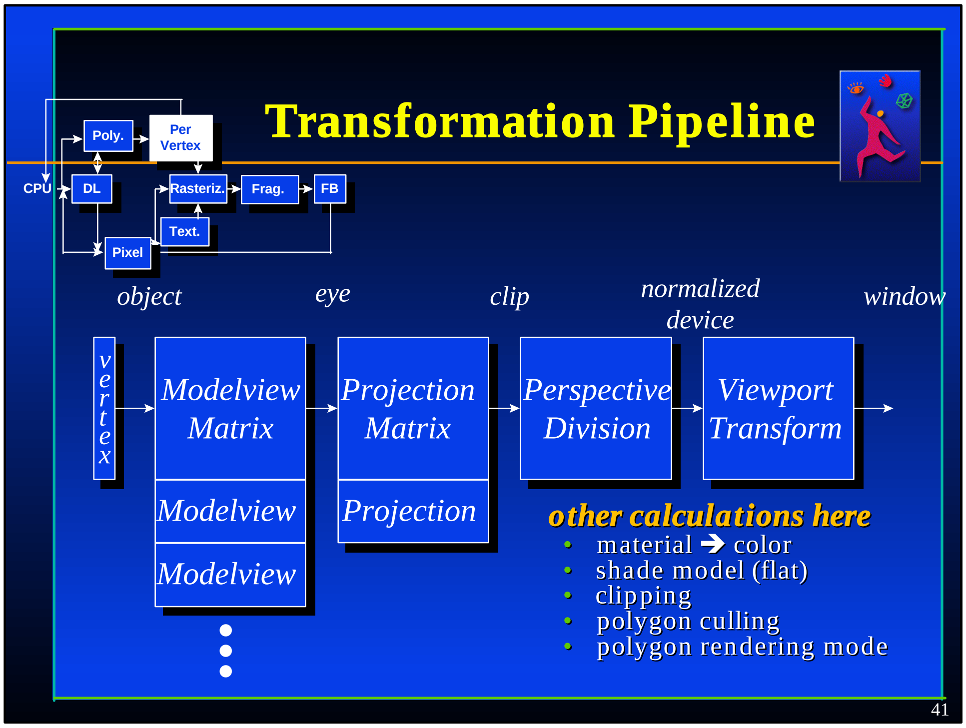


A 3D vertex is represented by a 4-tuple vector (homogeneous coordinate system).

Why is a 4-tuple vector (and a 4 by 4 matrix) used for a 3D (x, y, z) vertex? To ensure that all matrix operations are multiplications. Perspective projection and translation require 4th row and column, or operations would require addition, as well as multiplication.



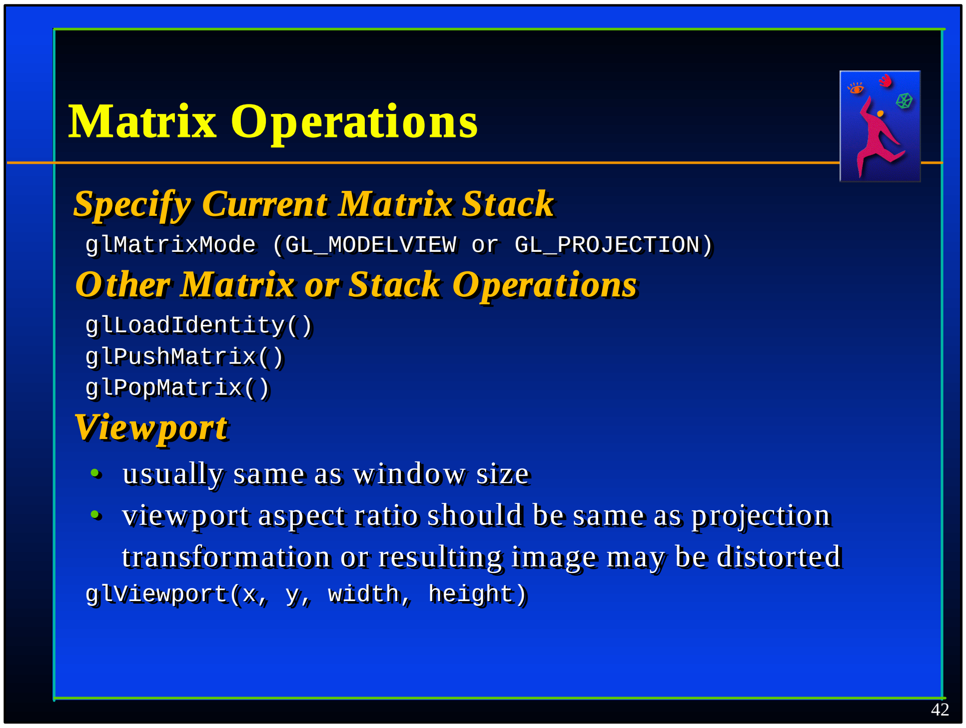
Note that human vision and a camera lens have cone-shaped viewing volumes. OpenGL (and almost all computer graphics APIs) describe a pyramid-shaped viewing volume. Therefore, the computer will “see” differently from the natural viewpoints, especially along the edges of viewing volumes. This is particularly pronounced for wide-angle “fish-eye” camera lenses.



The depth of matrix stacks are implementation-dependent, but the Modelview matrix stack is guaranteed to be at least 32 matrices deep, and the Projection matrix stack is guaranteed to be at least 2 matrices deep.

The material-to-color, flat-shading, and clipping calculations take place after the Modelview matrix calculations, but before the Projection matrix. The polygon culling and rendering mode operations take place after the Viewport operations.

There is also a texture matrix stack, which is not mentioned here. It is an advanced texture mapping topic.

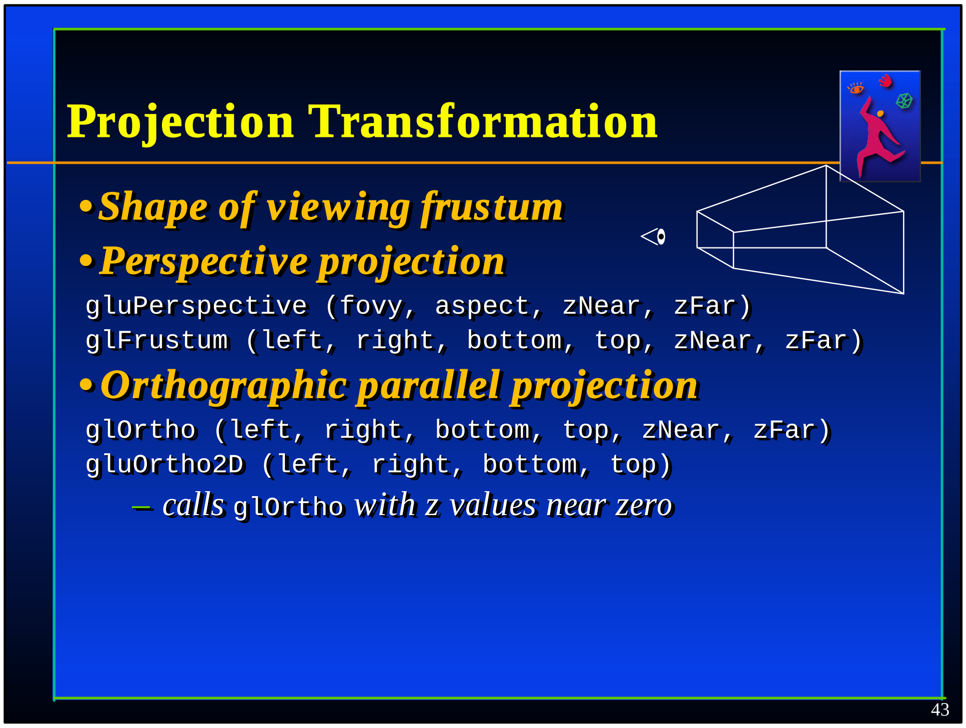


Also `glLoadMatrix{df}(matrix)` and `glMultMatrix{df}(matrix)`, where matrix is a column-major 4 x 4 double or single precision floating point matrix. The matrix is either loaded or post-multiplied onto the top of the current matrix stack.

The stacks are used, because it is more efficient to save and restore matrices than to calculate and multiply new matrices. Popping a matrix stack can be said to "jump back" to a previous location or orientation.

`glViewport` clips the vertex or raster position. For geometric primitives, a new vertex may be created. For raster primitives, the raster position is completely clipped.

There is a per-fragment operation, the scissor test, which works in situations where viewport clipping doesn't. The scissor operation is particularly good for fine clipping raster (bitmap or image) primitives.

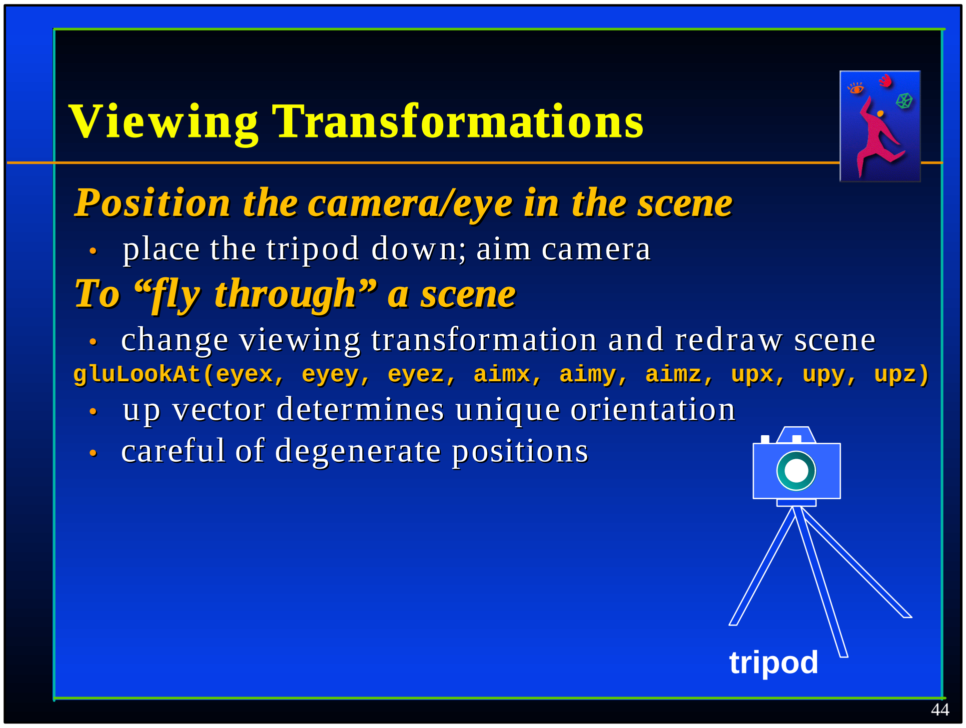


For perspective projections, the viewing volume is shaped like truncated pyramid (frustum). There is a distinct camera (eye) position, and vertexes of objects are "projected" to camera. Objects which are further from the camera appear smaller. The default camera position at (0, 0, 0), looking down z-axis, although the camera can be moved by other transformations.

For `gluPerspective`, `fovy` is the angle of field of view (in degrees) in the y direction. `fovy` must be between 0.0 and 180.0, exclusive. `aspect` is x/y and should be same as the viewport to avoid distortion `zNear` and `zFar` define the distance to the near and far clipping planes.

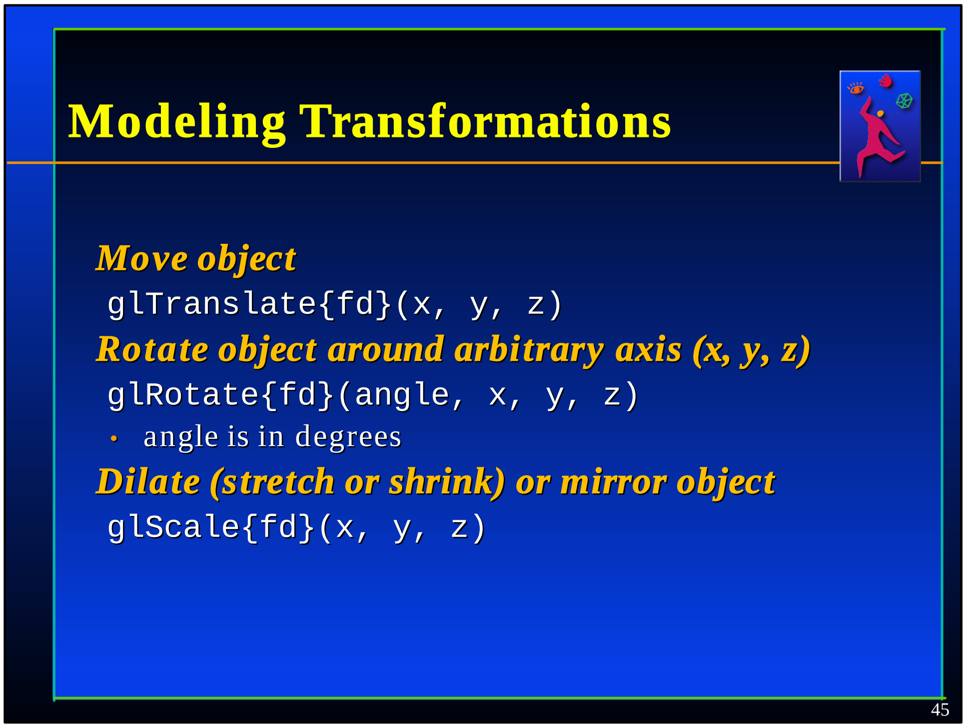
`glFrustum` is rarely used. Warning: for `gluPerspective` or `glFrustum`, don't use zero for `zNear`!

For `glOrtho()`, the viewing volume is shaped like a rectangular parallelepiped (a box). Vertexes of an object are "projected" towards infinity. Distance does not change the apparent size of an object. Orthographic projection is used for drafting and design (such as blueprints).



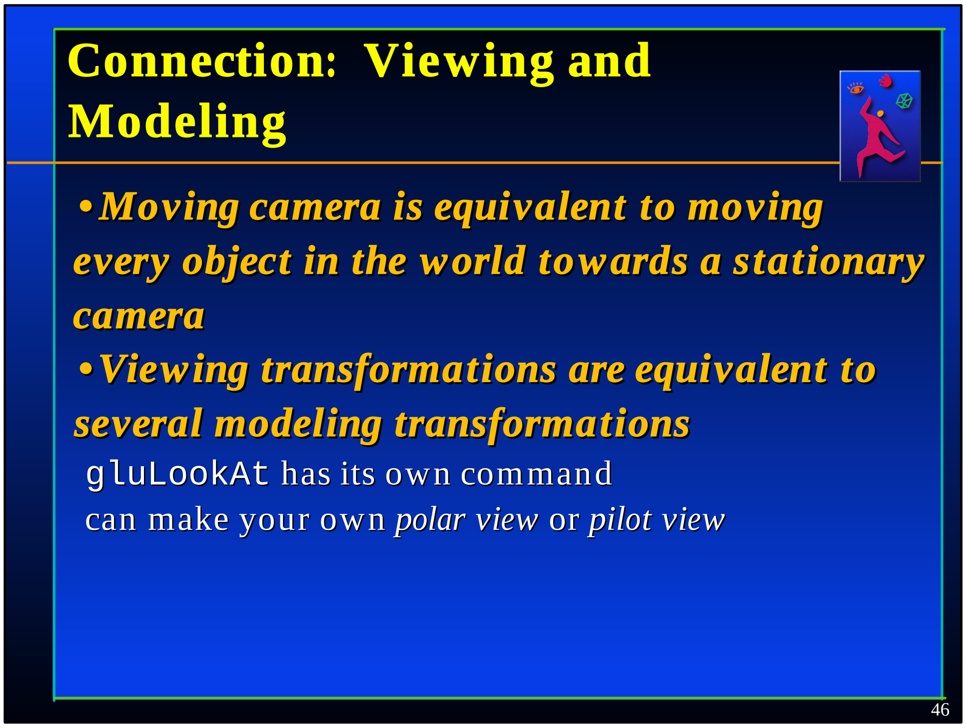
`gluLookAt()` multiplies itself onto the current matrix, so it usually comes after `glMatrixMode(GL_MODELVIEW)` and `glLoadIdentity()`.

Because of degenerate positions, `gluLookAt()` is not recommended for most animated fly-over applications.



`glTranslate()`, `glRotate()`, and `glScale()` multiplies itself onto the current matrix, so it usually comes after `glMatrixMode(GL_MODELVIEW)`. There are many situations where the modeling transformation is multiplied onto a non-identity matrix.

A vertex's distance from the origin changes the effect of `glRotate()` or `glScale()`. Generally, the further from the origin, the more pronounced the effect.



Instead of `gluLookAt`, one can use the following combinations of `glTranslate` and `glRotate` to achieve a viewing transformation. Like `gluLookAt`, these transformations should be multiplied onto the ModelView matrix, which should have an initial identity matrix.

To create a viewing transformation in which the viewer orbits an object, use this sequence (which is known as "polar view"):

```
glTranslated (0, 0, -distance)
glRotated (-twist, 0, 0, 1)
glRotated (-incidence, 1, 0, 0)
glRotated (azimuth, 0, 0, 1)
```

To create a viewing transformation which orients the viewer (roll, pitch, and heading) at position (x, y, z), use this sequence (known as "pilot view"):

```
glRotated (roll, 0, 0, 1)
glRotated (pitch, 0, 1, 0)
glRotated (heading, 1, 0, 0)
glTranslated (-x, -y, -z)
```

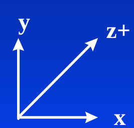
Projection is left handed

Projection transformations (`gluPerspective`, `glOrtho`) are left handed

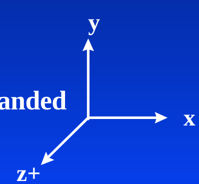
- think of `zNear` and `zFar` as distance from view point

Everything else is right handed, including the vertexes to be rendered

left handed



right handed



Common Transformation Usage

- ***3 examples of `resize()` routine***
  - restate projection & viewing transformations
- ***Usually called when window resized***
- ***Registered as callback for `glutReshapeFunc()`***

48

resize(): Perspective & LookAt

```
void resize (int w, int h) {
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective (65.0, (GLfloat) w/(GLfloat) h,
                   1.0, 100.0);
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0,
              0.0, 1.0, 0.0);
}
```

49

Using the viewport width and height as the aspect ratio for `gluPerspective` eliminates distortion.

resize(): Perspective & Translate

Same effect as previous LookAt

```
void resize (int w, int h) {
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective (65.0, (GLfloat) w/(GLfloat) h,
                   1.0, 100.0);
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
   glTranslatef (0.0, 0.0, -5.0);
}
```

50

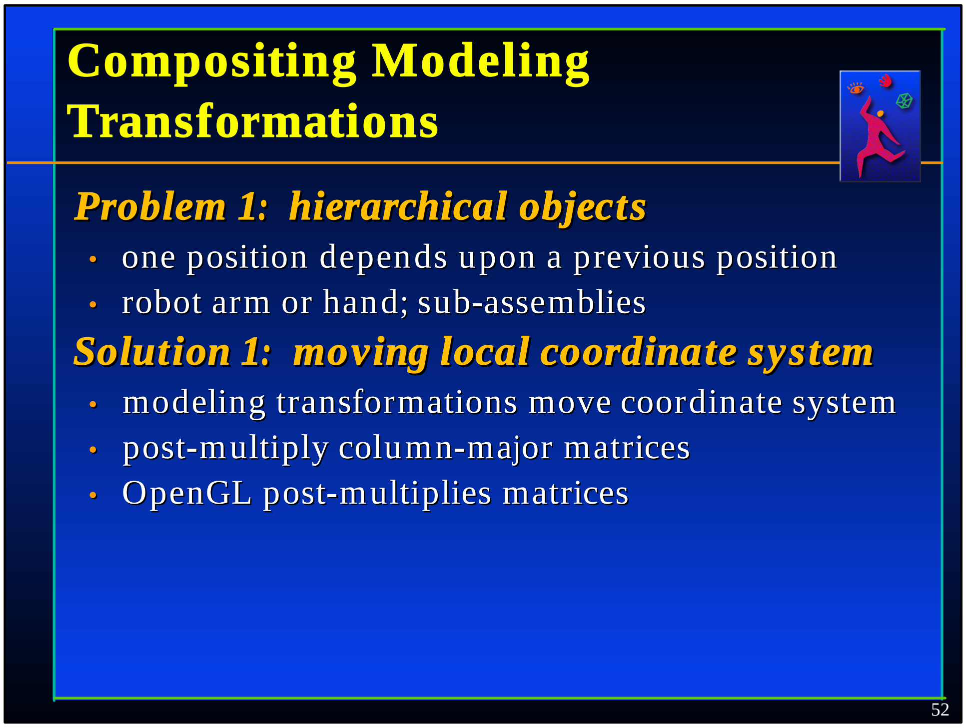
Moving all objects in the world five units away from you is mathematically the same as "backing up" the camera five units.

resize(): Ortho

```
void resize (int w, int h) {
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   if (w <= h)
      glOrtho (-2.5, 2.5, -2.5*(GLfloat)h/(GLfloat)w,
        2.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho (-2.5*(GLfloat)w/(GLfloat)h,
        2.5*(GLfloat)w/(GLfloat)h, -2.5, 2.5, -10.0, 10.0);
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
}
```

51

The two `glOrtho` calls are needed to accommodate for different aspect ratios, maintaining a minimum viewable region.



The order in which modeling transformations are performed is important because each modeling transformation is represented by a matrix, and matrix multiplication is not commutative. So a rotate followed by a translate is different from a translate followed by a rotate.

Compositing Modeling Transformations

Problem 2: objects move relative to absolute world origin

- my object rotates around the wrong origin
  - *make it spin around its center or something else*

Solution 2: fixed coordinate system

- modeling transformations move objects around fixed coordinate system
- pre-multiply column-major matrices
- OpenGL post-multiplies matrices
- must reverse order of operations to achieve desired effect

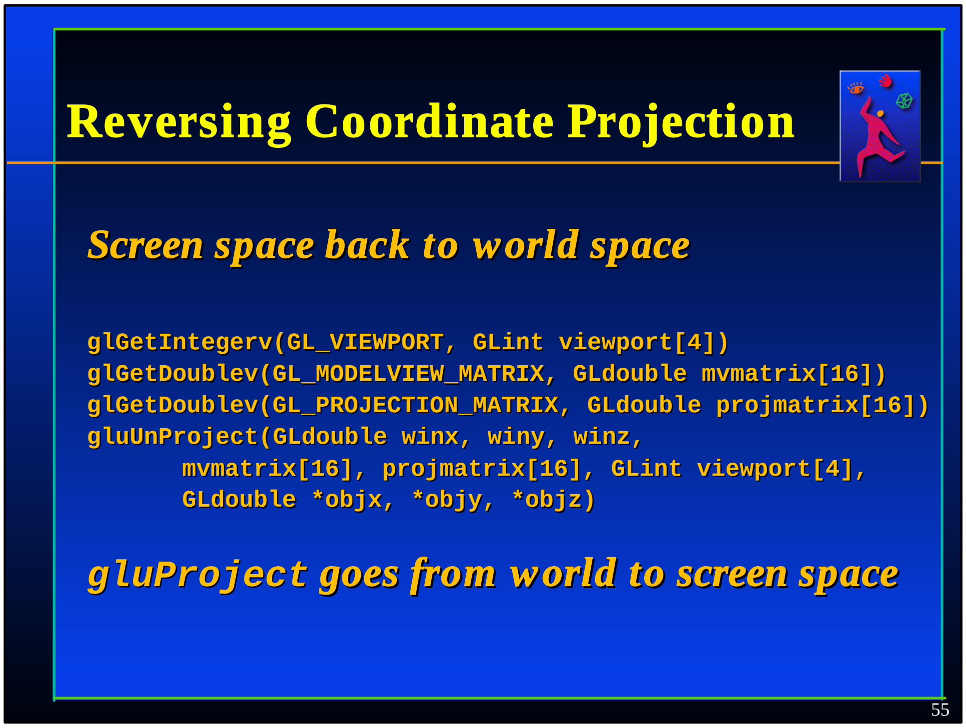
53

You'll adjust to reading a lot of code backwards!

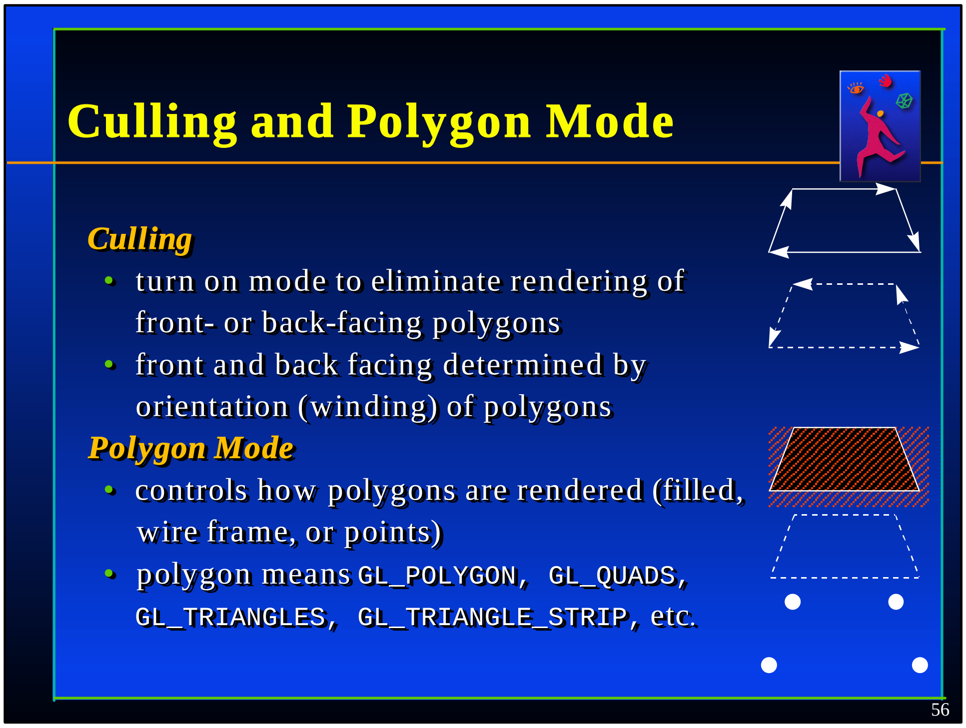
Additional Clipping Planes

- *At least 6 more clipping planes available*
- *Good for cross-sections*
- *Modelview matrix moves clipping plane*
- $Ax+By+Cz+D < 0$ *clipped*

```
glEnable (GL_CLIP_PLANEi)
glClipPlane (GL_CLIP_PLANEi, GLdouble* coeff)
```



Generally, OpenGL projects 3D data onto a 2D screen. Sometimes, you need to use a 2D screen position (such as a mouse location) and figure out where in 3D it came from. If you use `gluUnProject` with `winz` = 0 and `winz` = 1, you can find the 3D point at the near and far clipping planes. Then you can draw a line between those points, and you know that some point on that line was projected to your screen position.



`glFrontFace (windingMode)` determines which winding method is front- and back-facing for both culling and polygon mode. `windingMode` is `GL_CCW` or `GL_CW` (counter clockwise or clockwise). `GL_CCW` is the default.

OpenGL routines for culling include:

```
glEnable (GL_CULL_FACE)
glCullFace (whichWay)
```

`whichWay` is `GL_FRONT` or `GL_BACK` (the default)

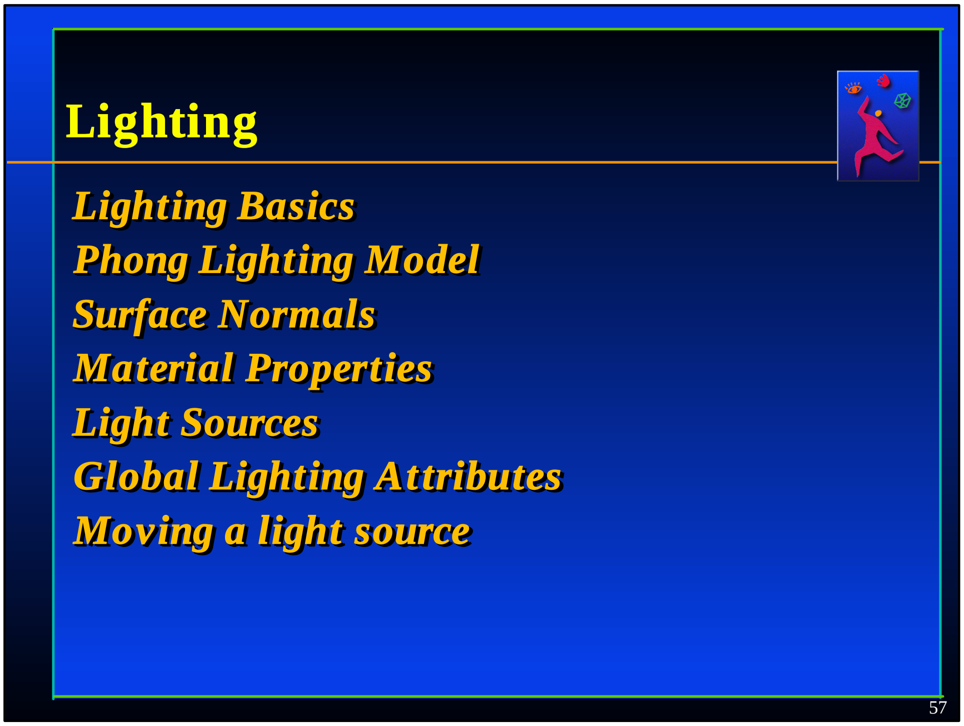
The application (and associated data sets) are responsible for creating the proper winding of the polygons. Culling is disabled by default.

The OpenGL routine for selecting polygon mode is:

```
glPolygonMode (GLenum face, GLenum mode)
```

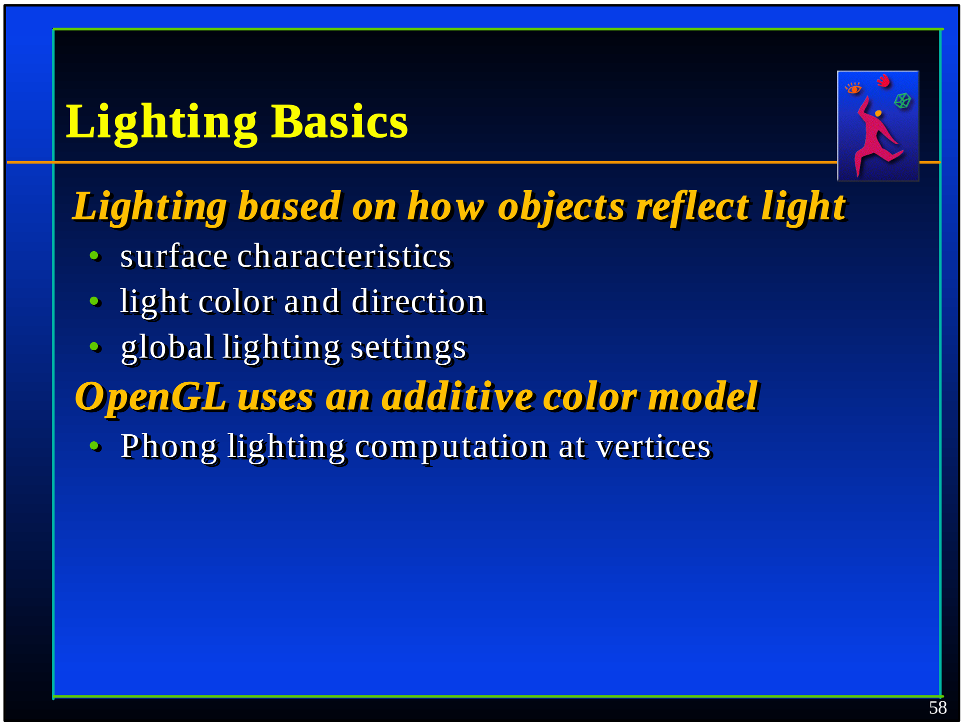
where face is GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK

mode is GL_FILL (default), GL_LINE, or GL_POINT



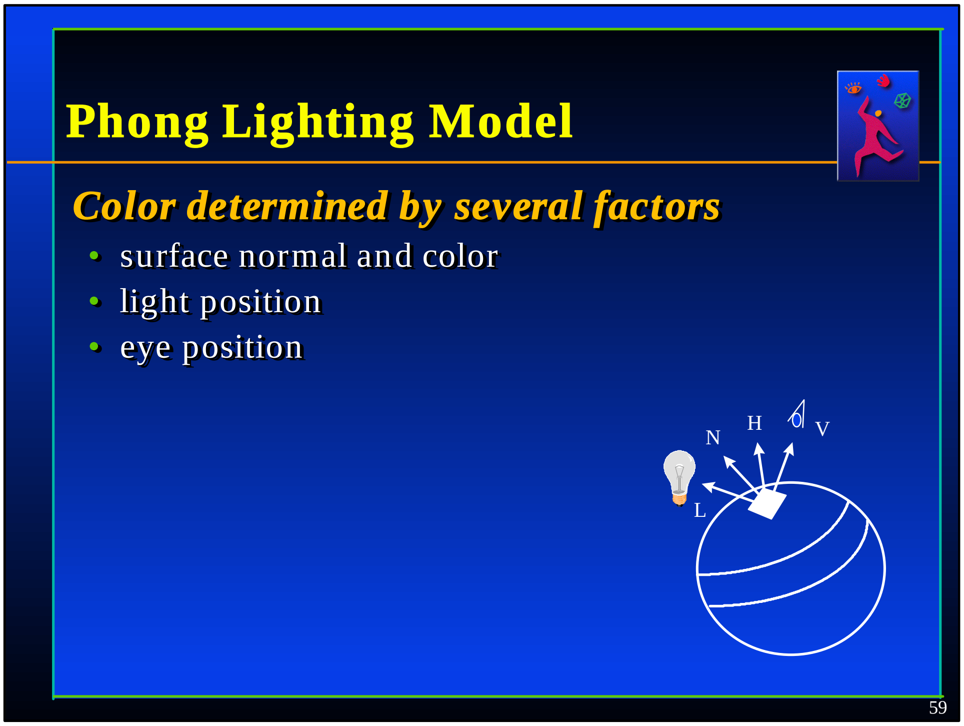
In this section we discuss OpenGL Lighting, which is based on the Phong lighting model.

The lighting model combines an object’s material properties and position, the light’s color and position, and global lighting parameters to compute a vertex’s color.



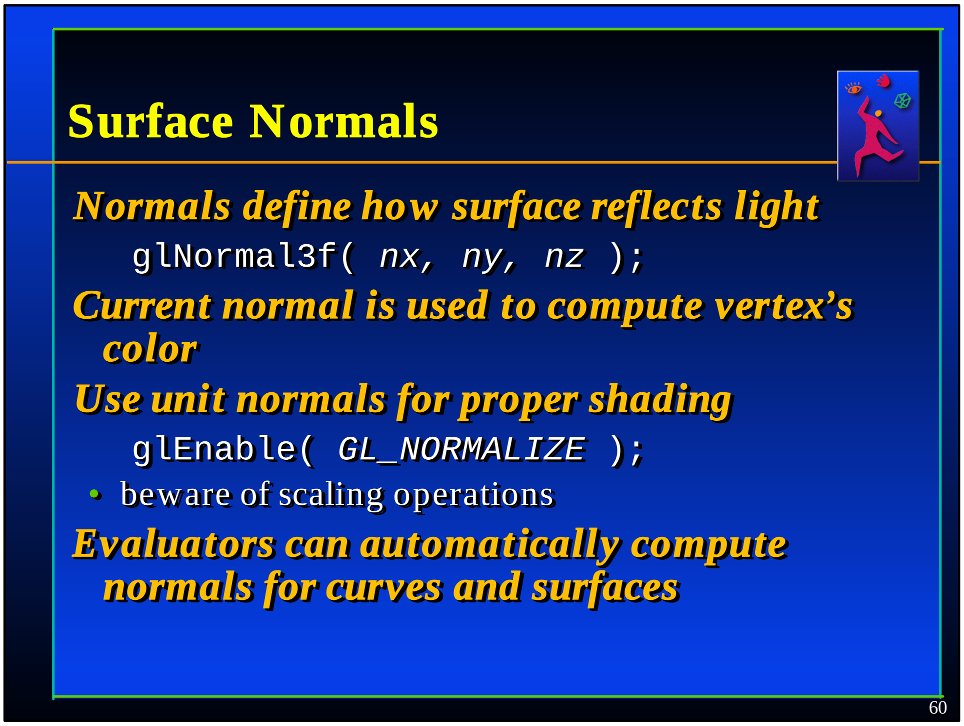
OpenGL uses a Phong based lighting model to compute colors for vertices. The color generated is a combination of the object’s material properties, the light’s color and position, and the global lighting characteristics.

Since lighting is used to compute a vertex’s color, the tessellation of the object is important. The better the tessellation and lighting normals, the better the lighting effects generated.



OpenGL uses an approximation to the Phong lighting model to compute the color of a lit vertex. In the diagram above, the following vectors are shown:

- V - vector from eyepoint to vertex
- N - vertex lighting normal
- L - vector from vertex to light
- H - 1/2 (V + L)

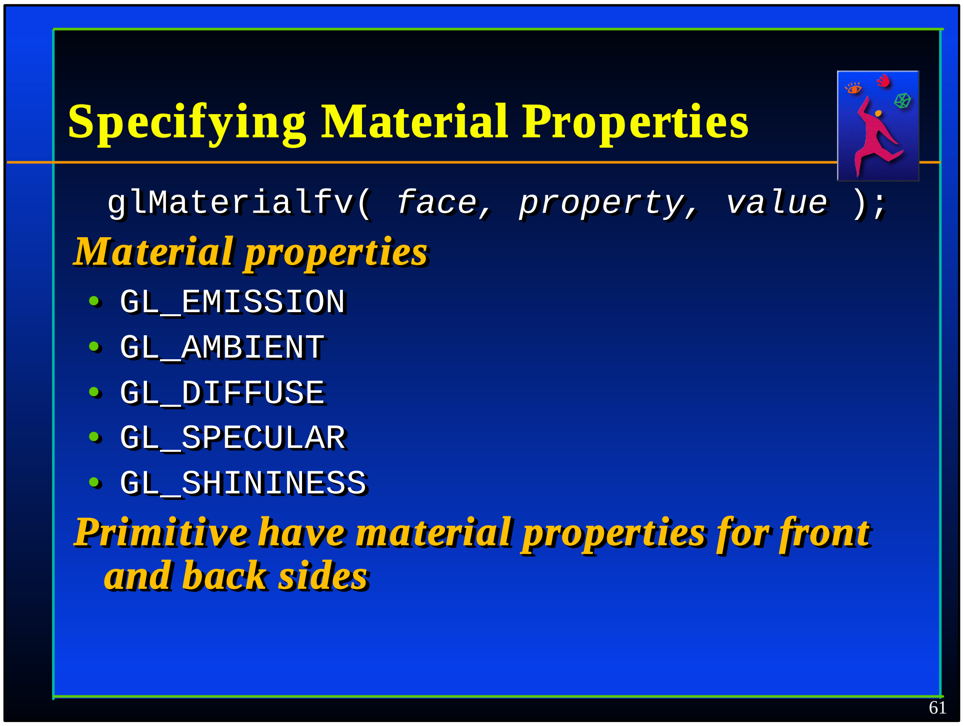


The lighting normal tells OpenGL how the object reflects light around a vertex. If you imagine that there is a small mirror at the vertex, the lighting normal describes how the mirror is oriented, and consequently how light is reflected.

`glNormal()` sets the current normal, which is used in the lighting computation for all vertices until a new normal is provided.

Lighting normals should be normalized to unit length for correct lighting results. `glScale()` affects normals as well as vertices, which can change the normal's length, and cause it to no longer be normalized. OpenGL can automatically normalize normals, by enabling `glEnable(GL_NORMALIZE)`. Since normalization requires the computation of a square root, it can potentially lower performance.

OpenGL evaluators and NURBS can provide lighting normals for generated vertices automatically.

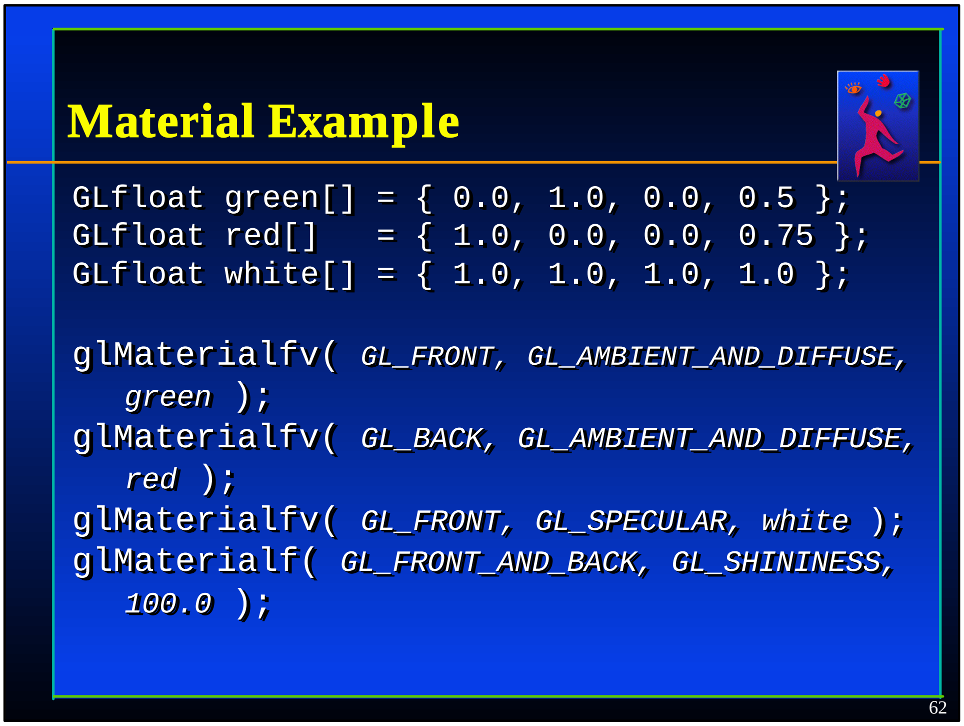


Material properties describe the color and surface properties of a material (dull, shiny, etc.). OpenGL supports material properties for both the front and back of objects, as described by their vertex winding.

The OpenGL material properties are:

- `GL_EMISSION` - color emitted from the object (think of a firefly)
- `GL_AMBIENT` - color of object when not directly illuminated
- `GL_DIFFUSE` - base color of object
- `GL_SPECULAR` - color of highlights on object
- `GL_SHININESS` - concentration of highlights on objects. Values range from 0 (very rough surface - no highlight) to 128 (very shiny)

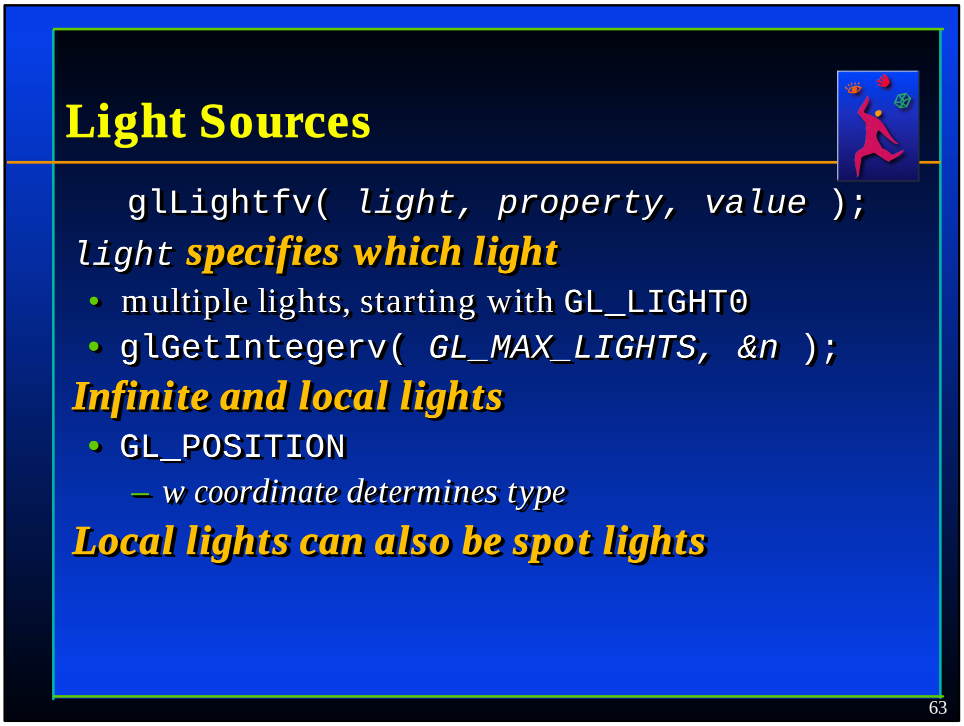
Material properties can be set for each face separately by specifying either `GL_FRONT` or `GL_BACK`, or for both faces simultaneously using `GL_FRONT_AND_BACK`.



In the above example, we set the following properties for the material:

- ambient and diffuse colors for the front side (as determined by the vertex winding) to a 50% opaque green
- ambient and diffuse color for the back side to a 75% opaque red color
- specular color for the front side to be full intensity white
- shininess to 100 (shiny) for both the front and back sides.

The transparency of an object is controlled only by the diffuse material's alpha value.

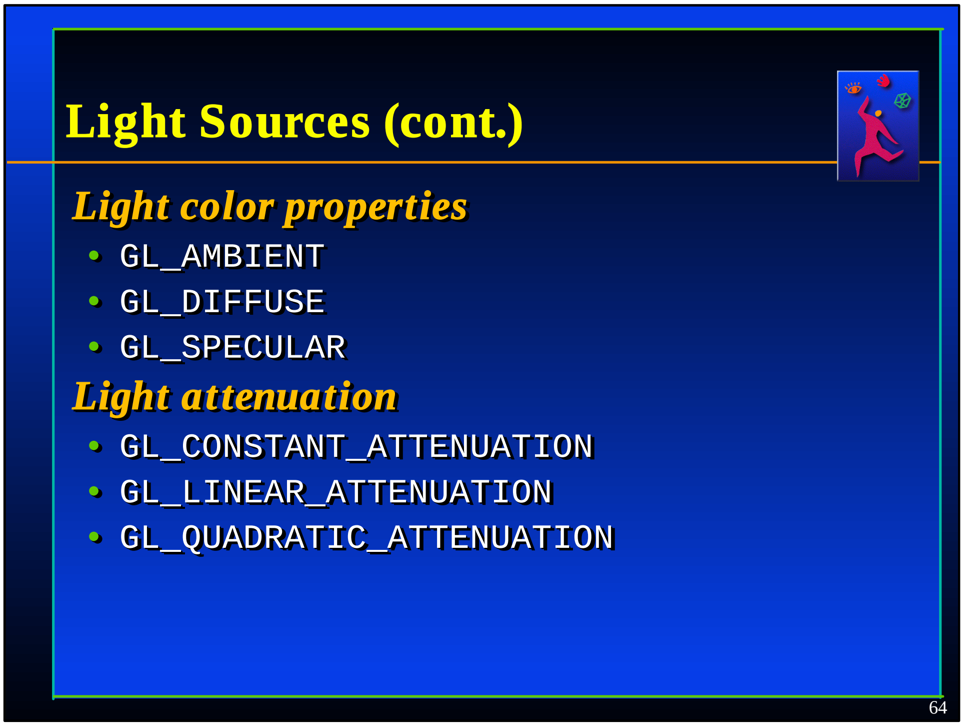


The `glLight()` call is used to set the parameters for a light. OpenGL implementations must support at least eight lights, which are named `GL_LIGHT0` through `GL_LIGHT`*n*, where *n* is one less than the maximum number supported by an implementation.

OpenGL supports two types of lights: infinite (directional) and local (point) light sources. The type of light is determined by the w coordinate of the light's position.

- if (w == 0.0) then the light is an infinite light, with (x,y,z) specifying the light's direction
- if (w != 0.0) then the light is a local light, with (x/w,y/w,z/w) specifying the light's position

A local light can also be converted into a spot light. By setting the `GL_SPOT_DIRECTION`, `GL_SPOT_CUTOFF`, and `GL_SPOT_EXPONENT`, the local light will shine in a direction and its light will be limited to a cone centered around that direction vector.



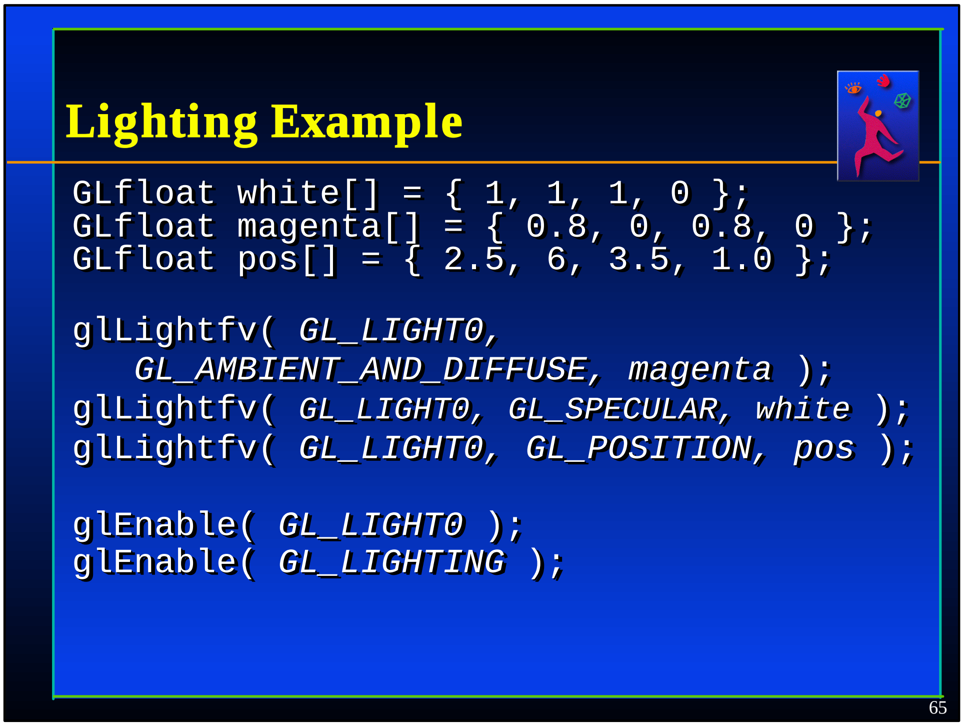
OpenGL light's can emit different colors for each of a materials properties. For example, a light's GL_AMBIENT color is combined with a material's GL_AMBIENT color to produce the ambient contribution to the color - Likewise for the diffuse and specular colors.

Each OpenGL light source supports attenuation, which describes how light diminishes with distance. The OpenGL model supports quadratic attenuation, and utilized the following attenuation factor, f_i, where d is the distance from the eyepoint to the vertex being lit:

$$f_i = \frac{1.0}{k_0 + k_1 * d + k_2 * d^2}$$

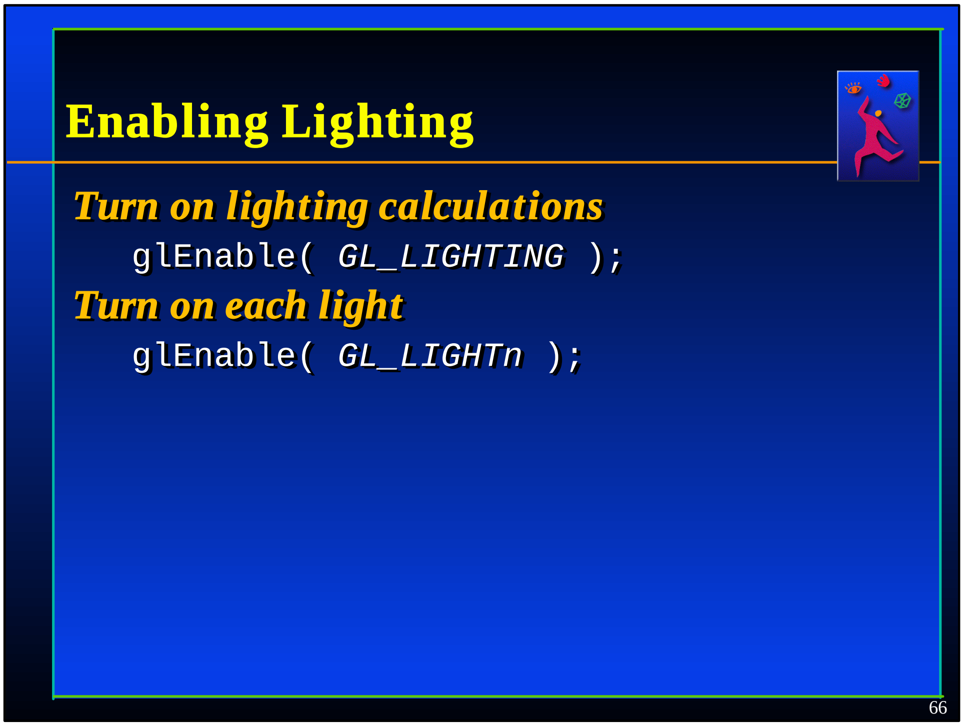
where:

- k_0 is the GL_CONSTANT_ATTENUATION term
- k_1 is the GL_LINEAR_ATTENUATION term
- k_2 is the GL_QUADRATIC_ATTENUATION term



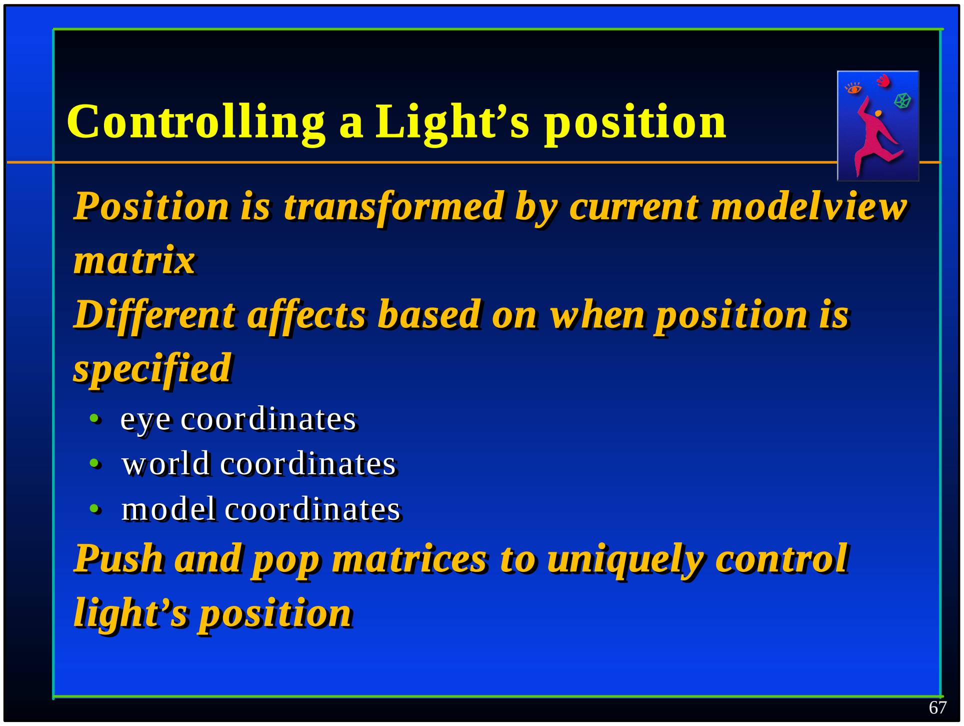
In the above example, we set the following properties for the light:

- ambient and diffuse colors to a shade of magenta
- specular color to pure white
- position to (2.5, 6, 3.5) and with w = 1.0, define the light to be a local light
- turn on both `GL_LIGHT0` and enable `GL_LIGHTING`



To enable the lighting computations, you need to enable each light which you would like to use in the scene, using `glEnable( GL_LIGHTn )`, where *n* represents the light.

Additionally, you need to globally enable lighting using `glEnable( GL_LIGHTING )`.

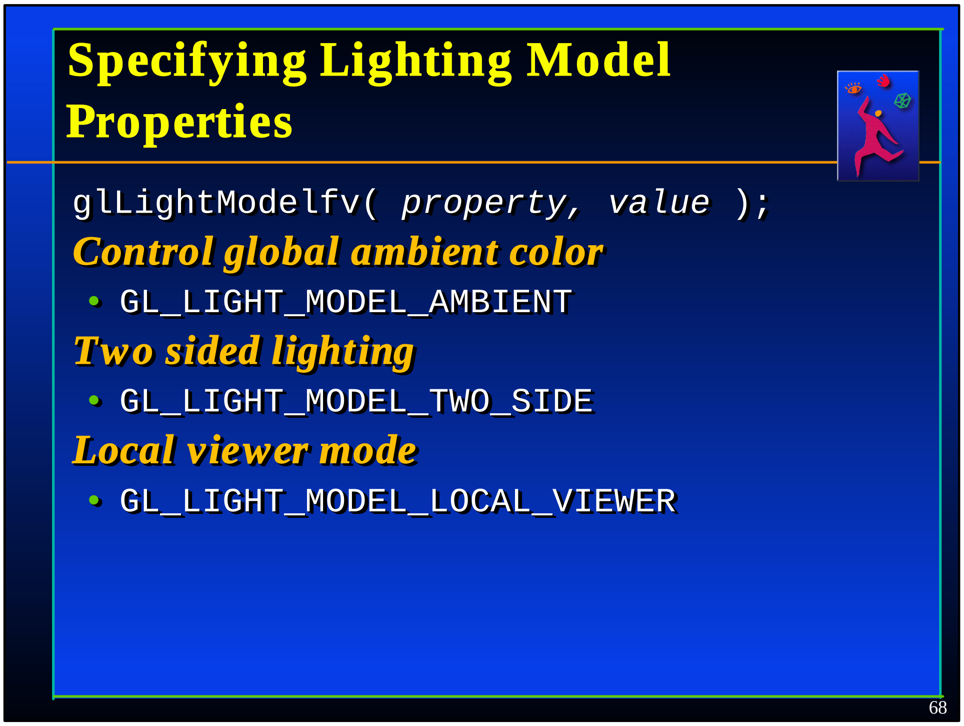


By specifying different ModelView transformations, you can achieve different types of lighting effects, since the light’s position is transformed when the `glLight()` call is issued.

If you specify the light’s position before any modeling or viewing transformations (eye coordinate space), you’ll achieve a headlamp effect, where the light’s emanating from the eyepoint.

If you specify after the viewing transformation, but before any modeling, the light’s position will remain fixed relative to the world coordinate system of the scene. This is like mounting the light on a steel rod at the origin. Regardless of how you move the world coordinates, the light’s position always remains constant relative to the world origin.

Finally, if you issue both modeling and viewing transforms, you can animate the light independent of the eye or anything else in your scene.

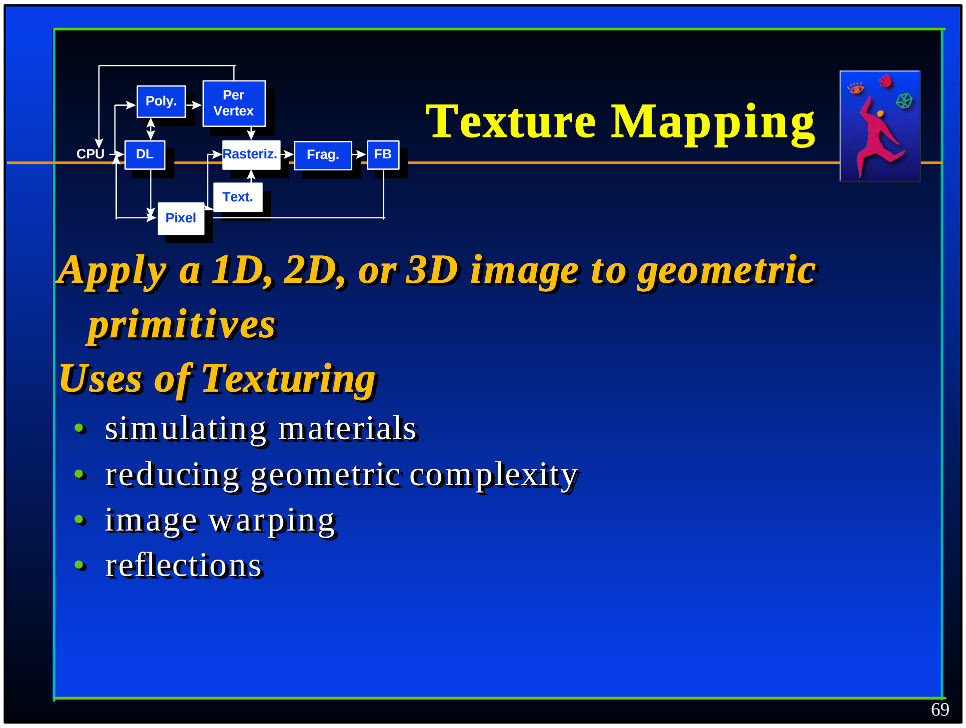


`glLightModel()` controls the global parameters of lighting such as the ambient light not contributed by a light and two sided lighting.

`GL_LIGHT_MODEL_AMBIENT` sets the global ambient color for the scene. This is used in combination with the material's ambient to produce ambient lighting, even if no lights are enabled.

`GL_LIGHT_MODEL_TWO_SIDE` is used to enable primitives to have different material properties for each side. Based on the winding of the primitive (set with `glCullFace()`), you can set up front and back materials for primitives.

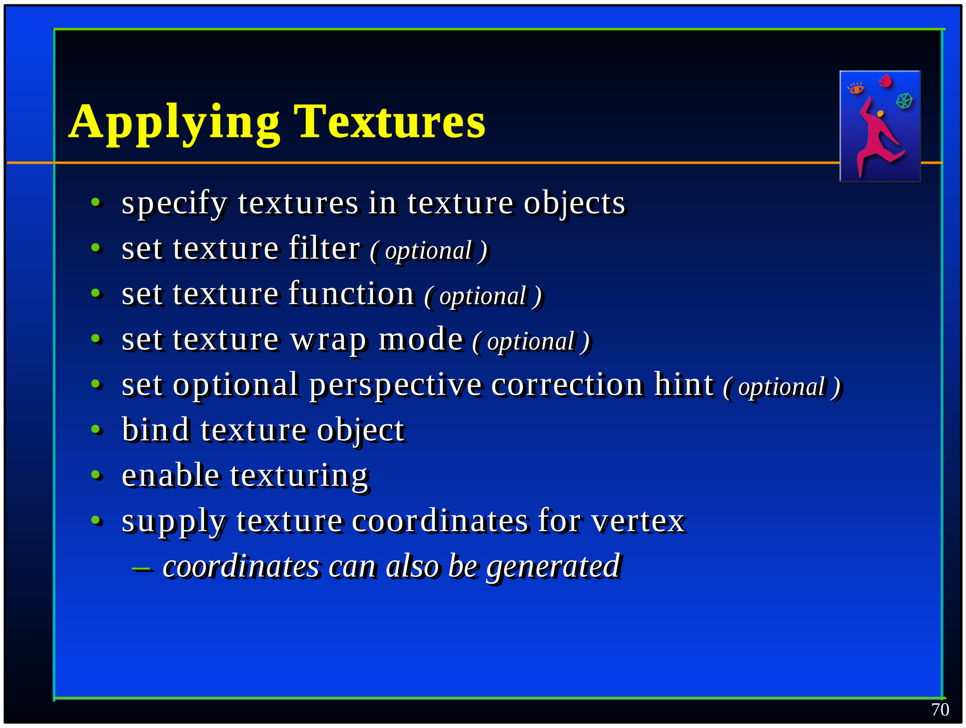
`GL_LIGHT_MODEL_LOCAL_VIEWER` is used to produce better lighting results, by eliminating some approximations made to make OpenGL lighting faster. This setting will produce better lighting results, but at a possible performance penalty.



In this section, we'll discuss *texture* (sometimes also called *image*) mapping. Texture mapping augments the colors specified for a geometric primitive with the colors stored in an image. An image can be a 1D, 2D, or 3D set of colors called *texels*. 2D textures will be used throughout the section for demonstrations, however, the processes described are identical for 1D and 3D textures.

Some of the many uses of texture mapping include:

- simulating materials like wood, bricks or granite
- reducing the complexity (number of polygons) of a geometric object
- image processing techniques like image warping and rectification, rotation and scaling
- simulating reflective surfaces like mirrors or polished floors



The general steps to enable texturing are listed above. Some steps are optional, and due to the number of combinations, complete coverage of the topic is outside the scope of this course.

We will be using the *texture object* approach. Using texture objects may enable your OpenGL implementation to make some optimizations behind the scenes.

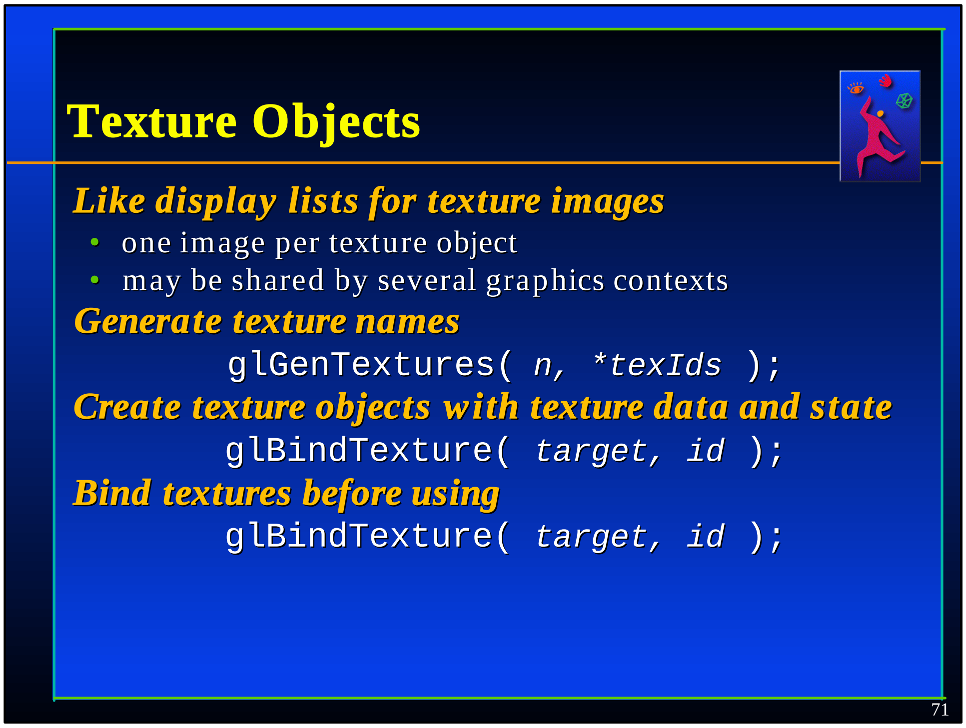
As with any other OpenGL state, texture mapping requires that `glEnable()` be called. The tokens for texturing are:

`GL_TEXTURE_1D` - one dimensional texturing

`GL_TEXTURE_2D` - two dimensional texturing

`GL_TEXTURE_3D` - three dimensional texturing

2D texturing is the most commonly used. 1D texturing is useful for applying contours to objects (like altitude contours to mountains). 3D texturing is useful for volume rendering.

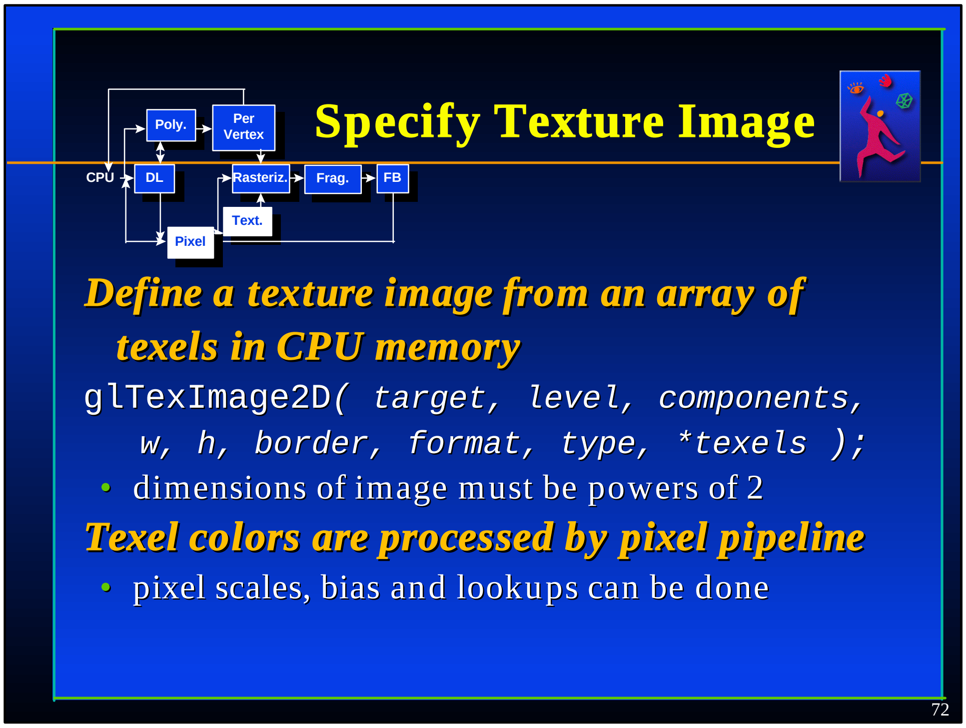


The first step in creating texture objects is to have OpenGL reserve some indices for your objects. `glGenTextures()` will request *n* texture ids and return those values back to you in *texIds*.

To begin defining a texture object, you call `glBindTexture()` with the *id* of the object you want to create. The *target* is on of `GL_TEXTURE_{123}D`. All texturing calls become part of the object until the next `glBindTexture()` is called.

To have OpenGL use a particular texture object, call `glBindTexture()` with the *target* and *id* of the object you want to be active.

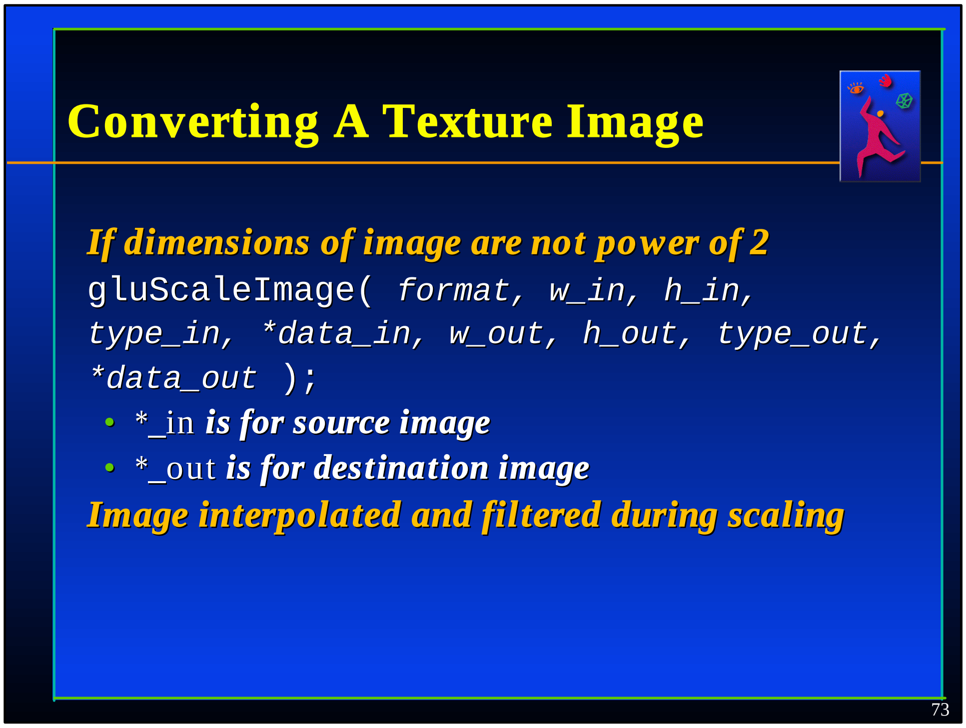
To delete texture objects, use `glDeleteTextures( n, *texIds )`, where *texIds* is an array of texture object identifiers to be deleted.



Specifying the texels for a texture is done using the `glTexImage{123}D()` call. This will transfer the texels in CPU memory to OpenGL, where they will be processed and converted into an internal format.

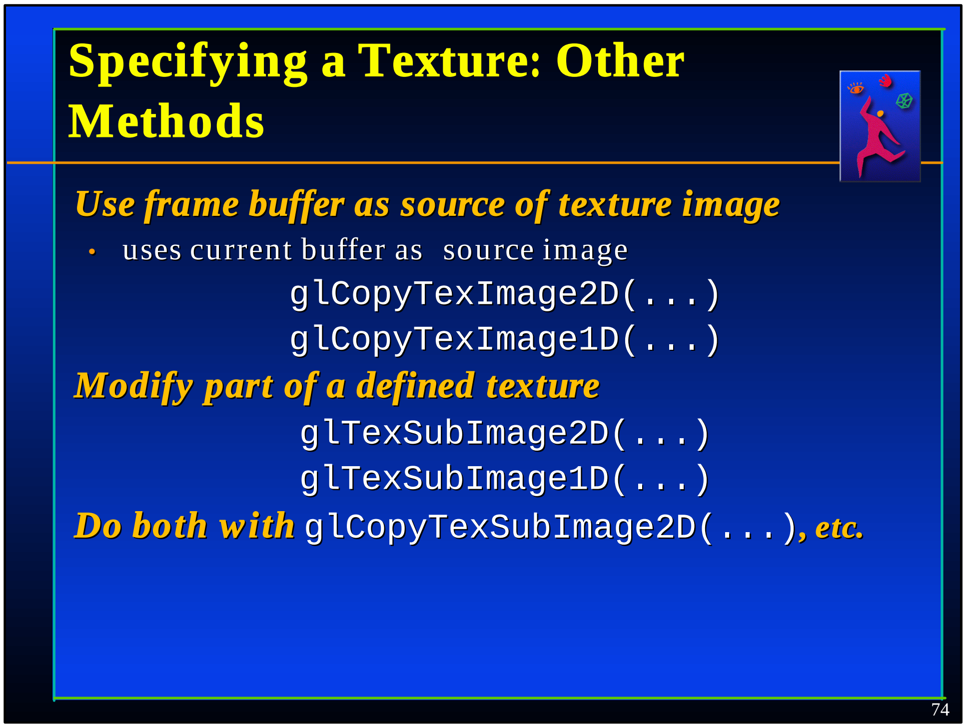
The array of texels sent to OpenGL with `glTexImage*()` must be a power of two in both directions. An optional one texel wide border may be added around the image. This is useful for certain wrapping modes.

The *level* parameter is used for defining how OpenGL should use this image when mapping texels to pixels. Generally, you'll set the *level* to 0, unless you're using a texturing technique called *mipmapping*, which we'll discuss in a few slides.



If your image does not meet the power of two requirement for a dimension, the `gluScaleImage()` call will resample an image to a particular size. It uses a simple box filter to interpolate the new images pixels from the source image.

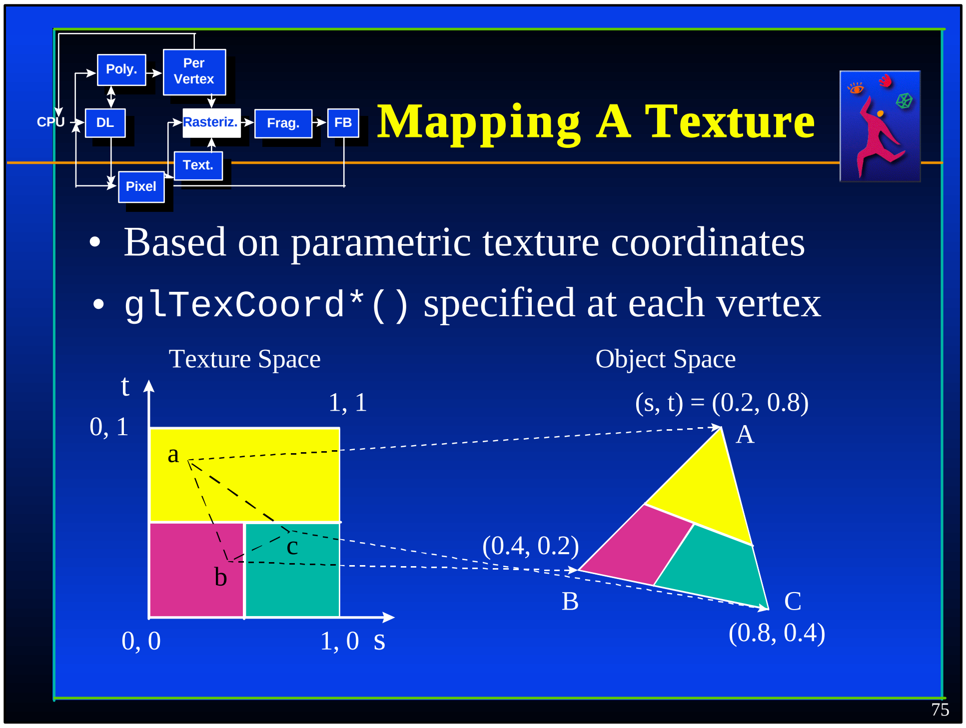
Additionally, `gluScaleImage()` can be used to convert from one data type (i.e. `GL_FLOAT`) to another type, which may better match the internal format in which OpenGL stores your texture.



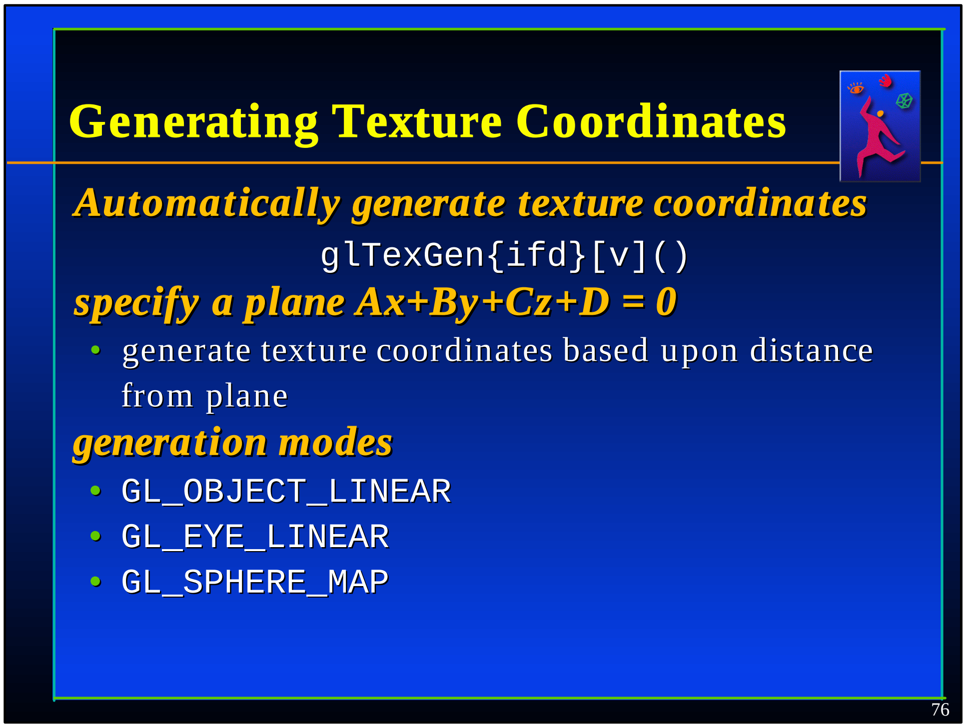
`glCopyTexImage*()` allows textures to be defined by in any of OpenGL's buffers. The source buffer is selected using the `glReadBuffer()` command.

Using `glTexSubImage*()` to replace all or part of an existing texture often outperforms using `glTexImage*()` to allocate and define a new one. This can be useful for creating a "texture movie" (sequence of textures which changes appearance on an object's surface).

There are some advanced techniques using `glTexSubImage*()` which include loading an image which doesn't meet the power of two requirement. Additionally, several small images can be "packed" into one larger image (which was originally created with `glTexImage*()`), and loaded individually with `glTexSubImage*()`. Both of these techniques require the manipulation of the texture transform matrix, which is outside the scope of this course.



When you want to map a texture onto a geometric primitive, you need to provide texture coordinates. The `glTexCoord*()` call sets the current texture coordinates. Valid texture coordinates are between 0 and 1, for each texture dimension, and the default texture coordinate is (0, 0, 0, 1). If you pass fewer texture coordinates than the currently active texture mode (for example, using `glTexCoord1d()` while `GL_TEXTURE_2D` is enabled), the additionally required texture coordinates take on default values.



You can have OpenGL automatically generate texture coordinates for vertices by using the `glTexGen()` and `glEnable( GL_TEXTURE_GEN_{STRQ} )`. The coordinates are computed by determining the vertex's distance from each of the enabled generation planes.

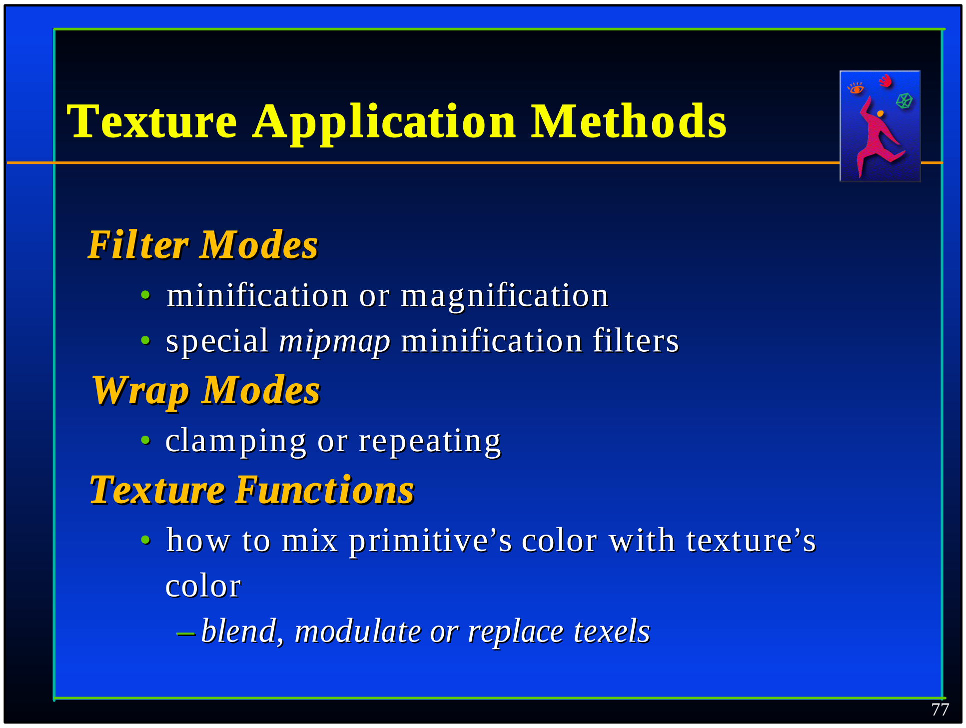
As with lighting positions, texture generation planes are transformed by the ModelView matrix, which allows different results based upon when the `glTexGen()` is issued.

There are three ways in which texture coordinates are generated:

`GL_OBJECT_LINEAR` - textures are fixed to the object (like wall paper)

`GL_EYE_LINEAR` - texture fixed in space, and object move through texture (like underwater light shining on a swimming fish)

`GL_SPHERE_MAP` - object reflects environment, as if it were made of mirrors (like the shiny guy in *Terminator 2*)



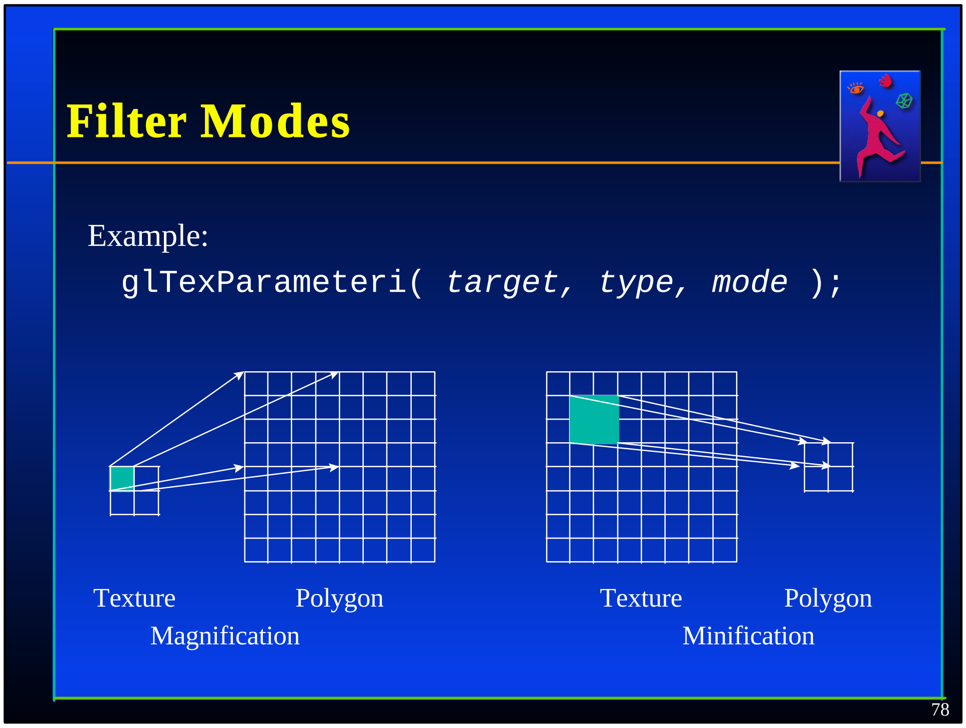
Textures and the objects being textured are rarely the same size (in pixels). *Filter modes* determine the methods used by how texels should be expanded (*magnification*), or shrunk (*minification*) to match a pixel's size. An additional technique, called *mipmapping* is a special instance of a minfication filter.

Wrap modes determine how to process texture coordinates outside of the [0,1] range. The available modes are:

`GL_CLAMP` - clamp any values outside the range to closest valid value, causing the edges of the texture to be "smeared" across the primitive

`GL_REPEAT` - use only the fractional part of the texture coordinate, causing the texture to repeat across an object

Finally, the texture environment describes how a primitives fragment colors and texel colors should be combined to produce the final framebuffer color. Depending upon the type of texture (i.e. intensity texture vs. RGBA texture) and the mode, pixels and texels may be simply multiplied, linearly combined, or the texel may replace the fragment's color altogether.



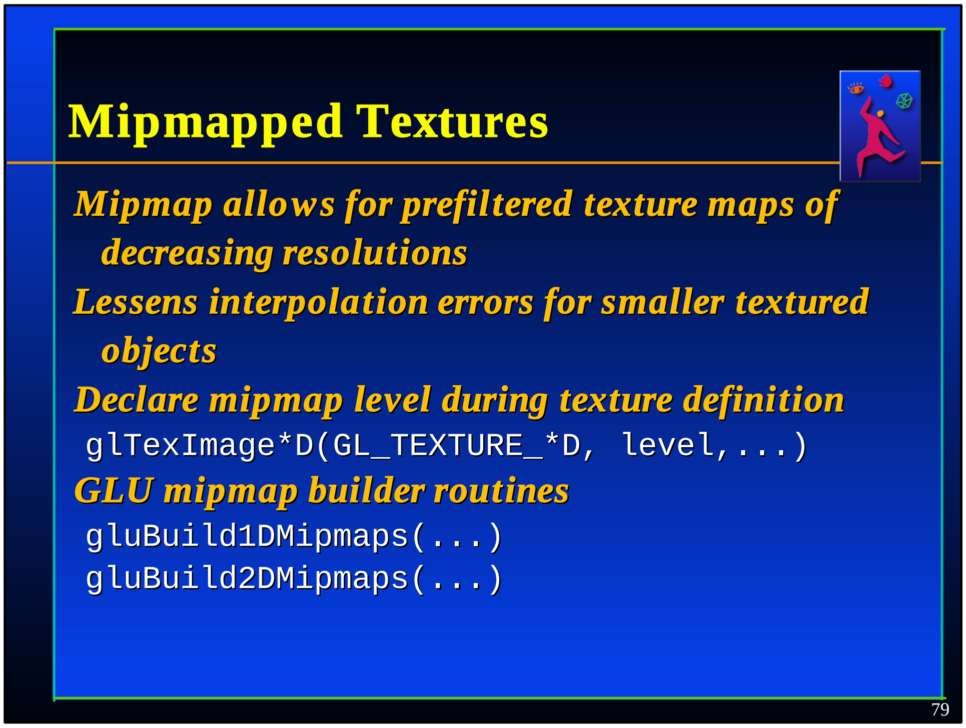
Filter modes control how pixels are minified or magnified. Generally a color is computed using the nearest texel or by a linear average of several texels.

The filter *type*, above is one of `GL_TEXTURE_MIN_FILTER` or `GL_TEXTURE_MAG_FILTER`.

The *mode* is one of `GL_NEAREST`, `GL_LINEAR`, or special modes for mipmapping. Mipmapping modes are used for minification only, and have values of:

```
GL_NEAREST_MIPMAP_NEAREST
GL_NEAREST_MIPMAP_LINEAR
GL_LINEAR_MIPMAP_NEAREST
GL_LINEAR_MIPMAP_LINEAR
```

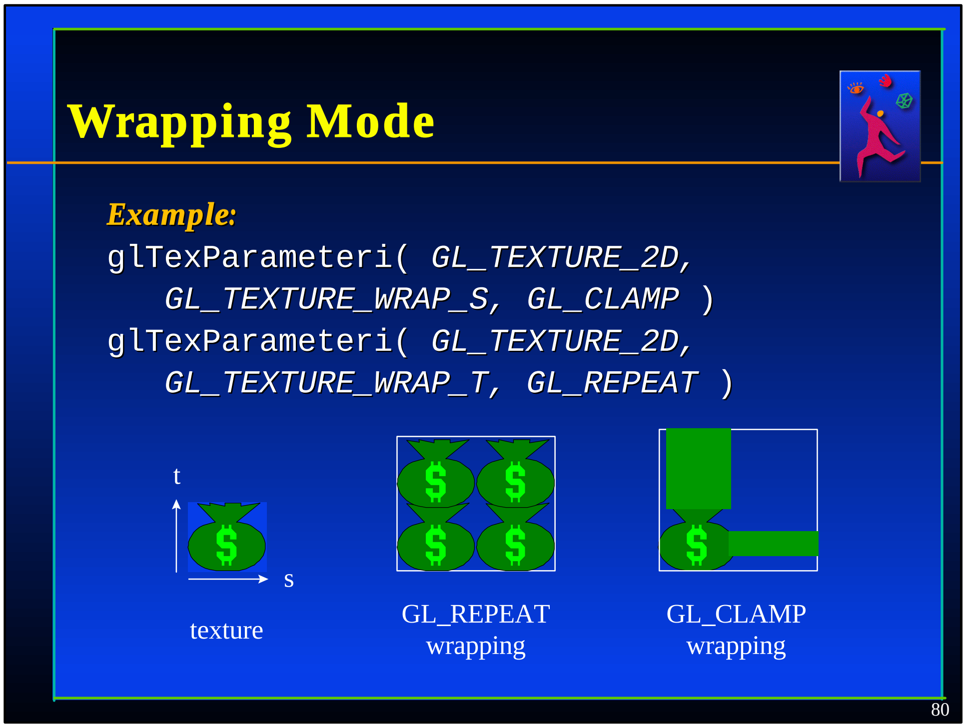
Full coverage of mipmap texture filters is outside the scope of this course.



As primitives become smaller in screen space, a texture may appear to shimmer as the minification filters creates rougher approximations. Mipmapping is an attempt to reduce the shimmer effect by creating several approximations to the original image at lower resolutions.

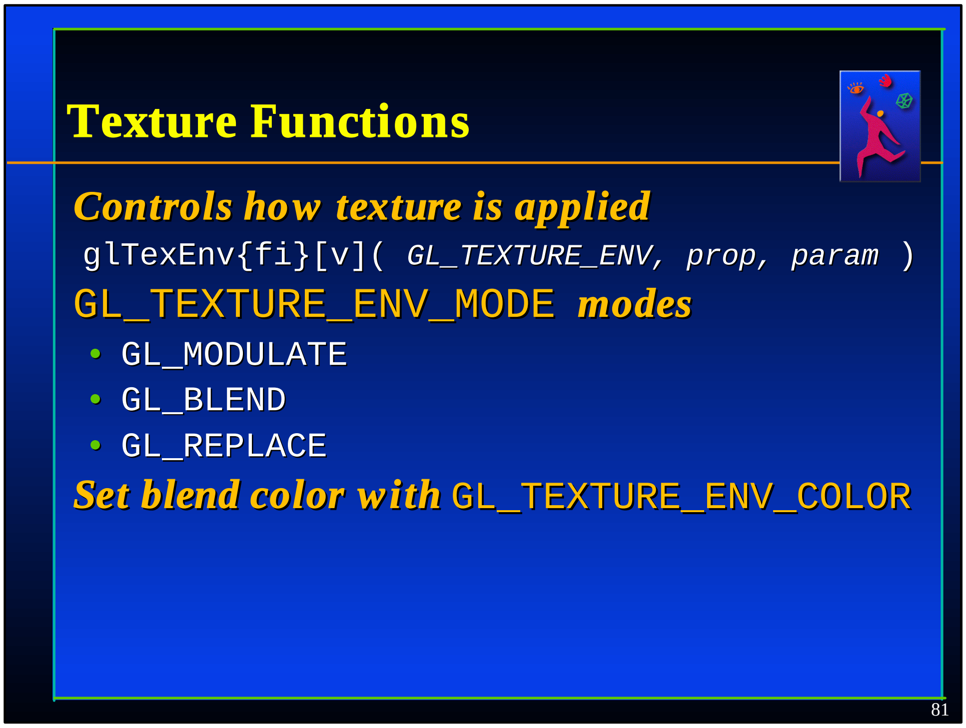
Each mipmap level should have an image which is one-half the height and width of the previous level, to an minimum of one texel in either dimension. For example, level 0 could be 32 x 8 texels. Then level 1 would be 16 x 4; level 2 would be 8 x 2; level 3, 4 x 1; level 4, 2 x 1; finally, level 5, 1 x 1.

The `gluBuild*Dmipmaps()` routines will automatically generate each mipmap image, and call `glTexImage*D()` with the appropriate *level* value.



Wrap mode determines what should happen if a texture coordinate lies outside of the [0,1] range. If the `GL_REPEAT` wrap mode is used, for texture coordinate values less than zero or greater than one, the integer is ignored and only the fractional value is used.

If the `GL_CLAMP` wrap mode is used, the texture value at the extreme (either 0 or 1) is used.



The texture mode determines how texels and fragment colors are combined. The most common modes are:

`GL_MODULATE` - multiply texel and fragment color

`GL_BLEND` - linearly blend texel, fragment, env color

`GL_REPLACE` - replace fragment's color with texel

If *prop* is `GL_TEXTURE_ENV_COLOR`, *param* is an array of four floating point values representing the color to be used with the `GL_BLEND` texture function.

Perspective Correction Hint

Texture coordinate and color interpolation

- either linearly in screen space
- or using depth / perspective values (slower)

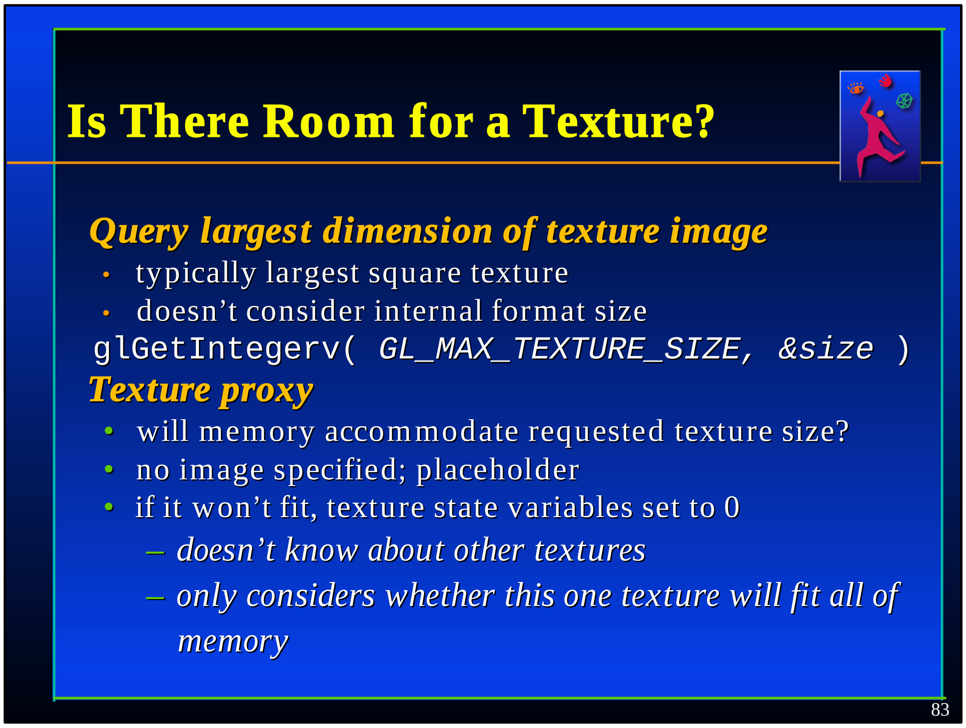
Noticeable for polygons "on edge"

`glHint(` *`GL_PERSPECTIVE_CORRECTION_HINT, hint`* `)`

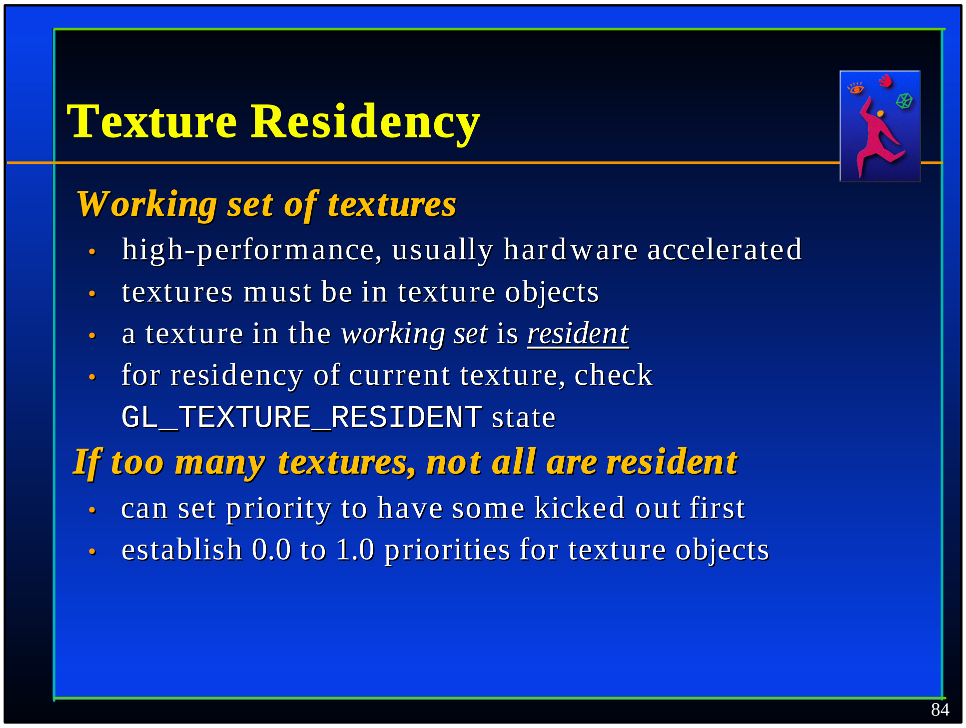
where `hint` is one of

- `GL_DONT_CARE`
- `GL_NICEST`
- `GL_FASTEST`

An OpenGL implementation may chose to ignore hints.




```
GLint proxyComponents;
glTexImage2D(GL_PROXY_TEXTURE_2D, 0, GL_RGBA8, 64, 64, 0,
GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D, 0,
GL_TEXTURE_COMPONENTS, &proxyComponents);
```



Query for residency of an array of texture objects:

```
GLboolean glAreTexturesResident(GLsizei n, GLuint *texNums,
GLboolean *residences)
```

Set priority numbers for an array of texture objects:

```
glPrioritizeTextures(GLsizei n, GLuint *texNums, GLclampf
*priorities)
```

Lower priority numbers mean that, in a crunch, these texture objects will be more likely to be moved out of the working set.

One common strategy is avoid prioritization, because many implementations will automatically implement an LRU (least recently used) scheme, when removing textures from the working set.

If there is no high-performance working set, then <u>all</u> texture objects are considered to be resident.

Overview of Other Topics

Display Lists

Feedback

- Picking / Selection

Image Primitives

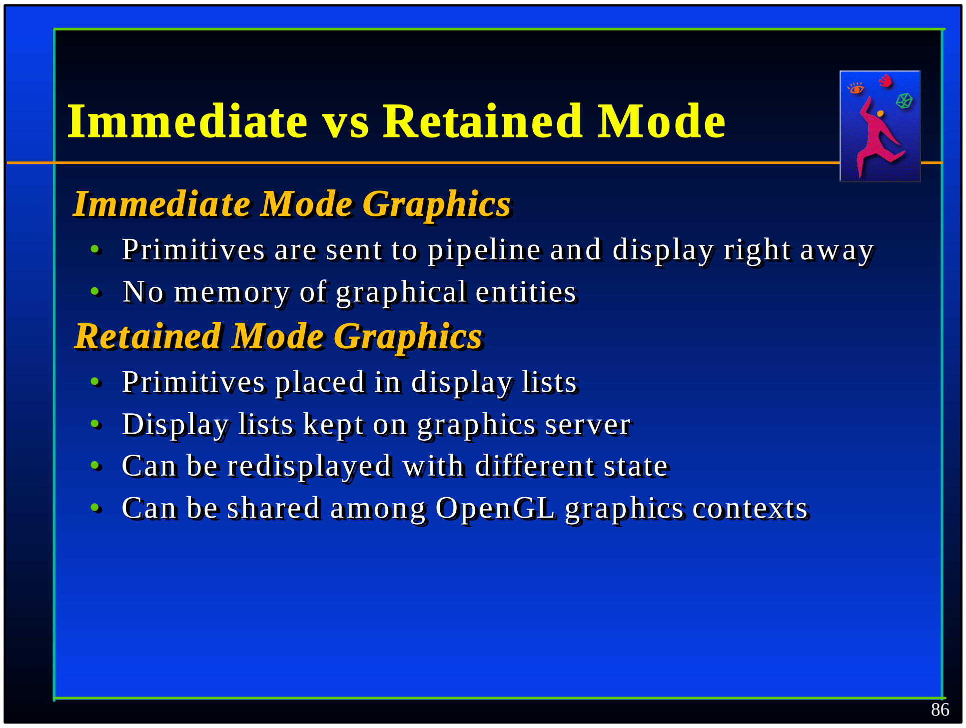
Fog

Per Fragment Operations

Blending

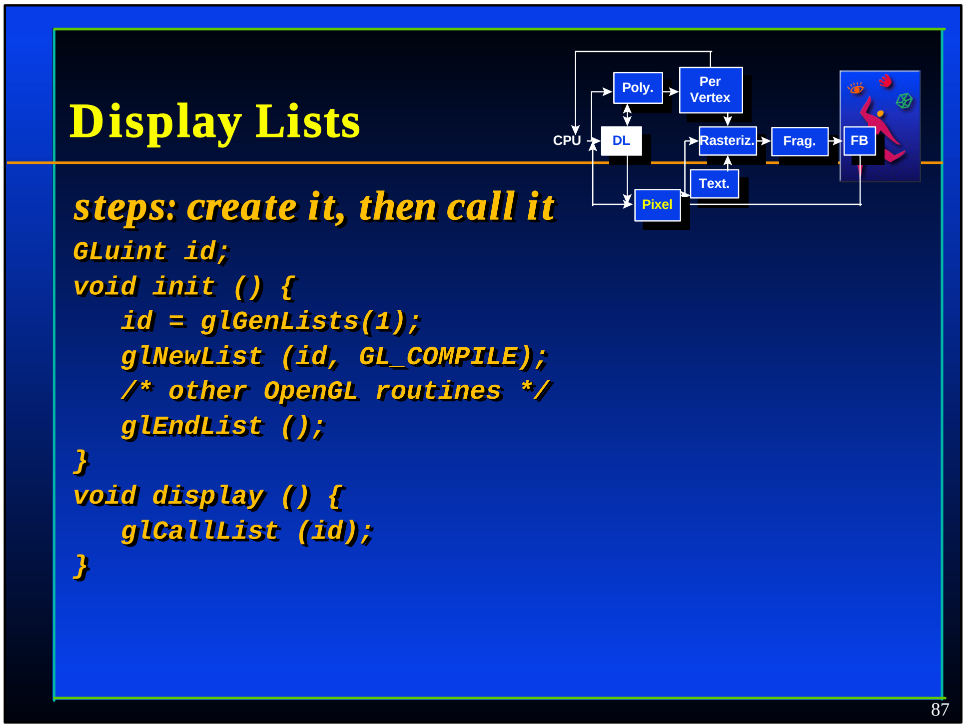
Antialiasing

85



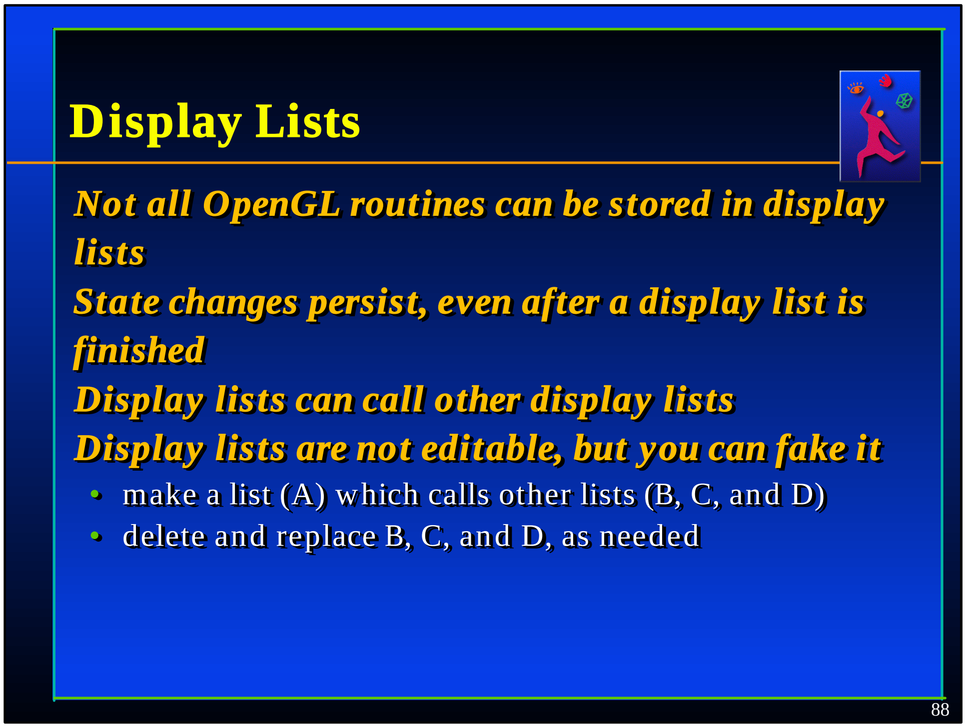
If display lists are shared, texture objects are also shared.

To share display lists among graphics contexts in the X Window System, use the `glXCreateContext` routine.



Instead of GL_COMPILE, `glNewList` also accepts the constant GL_COMPILE_AND_EXECUTE, which both creates and executes a display list.

If a new list is created with the same identifying number as an existing display list, the old list is deleted. No error occurs.



Some routines cannot be stored in a display list. Here is a list of them:

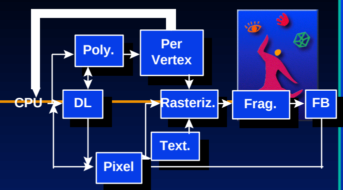
all `glGet*` routines

`glIs*` routines `(e.g., glIsEnabled, glIsList, glIsTexture)`

`glGenLists`	`glDeleteLists`	`glFeedbackBuffer`
`glSelectBuffer`	`glRenderMode`	`glVertexPointer`
`glNormalPointer`	`glColorPointer`	`glIndexPointer`
`glTexCoordPointer`		`glEdgeFlagPointer`
`glEnableClientState`		`glDisableClientState`
`glReadPixels`	`glPixelStore`	`glGenTextures`
`glDeleteTextures`		`glAreTexturesResident`
`glFlush`	`glFinish`	

If there is an attempt to store any of these routines in a display list, the routine is executed in immediate mode. No error occurs.

Feedback & Selection

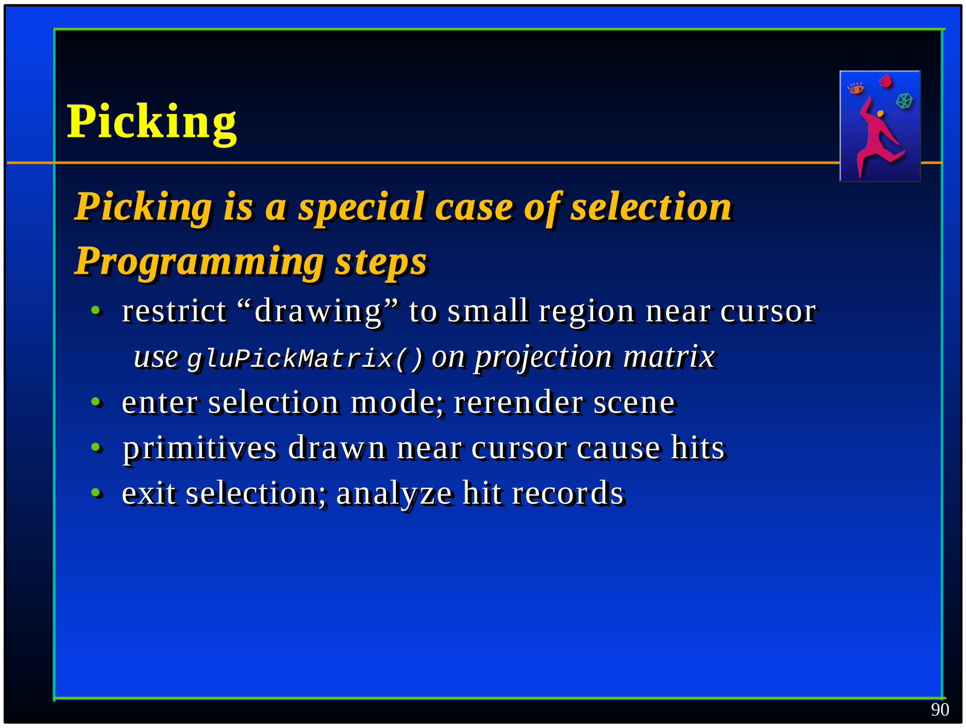


Usually, transformed vertices and colors generate an image in the frame buffer

Feedback mode: transformed values are returned to the application

Selection mode: if primitives are drawn within viewing volume, names are returned to the application

- in both cases, drawing stopped; no pixels are produced



The picking region is usually specified in a piece of code like this:

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
gluPickMatrix(x, y, width, height, viewport);
gluPerspective(...) or glOrtho(...)
```

The picking matrix is the rare situation where the standard projection matrix (perspective or ortho) is multiplied onto a non-identity matrix.

Each *hit record* contains:

number of names per hit

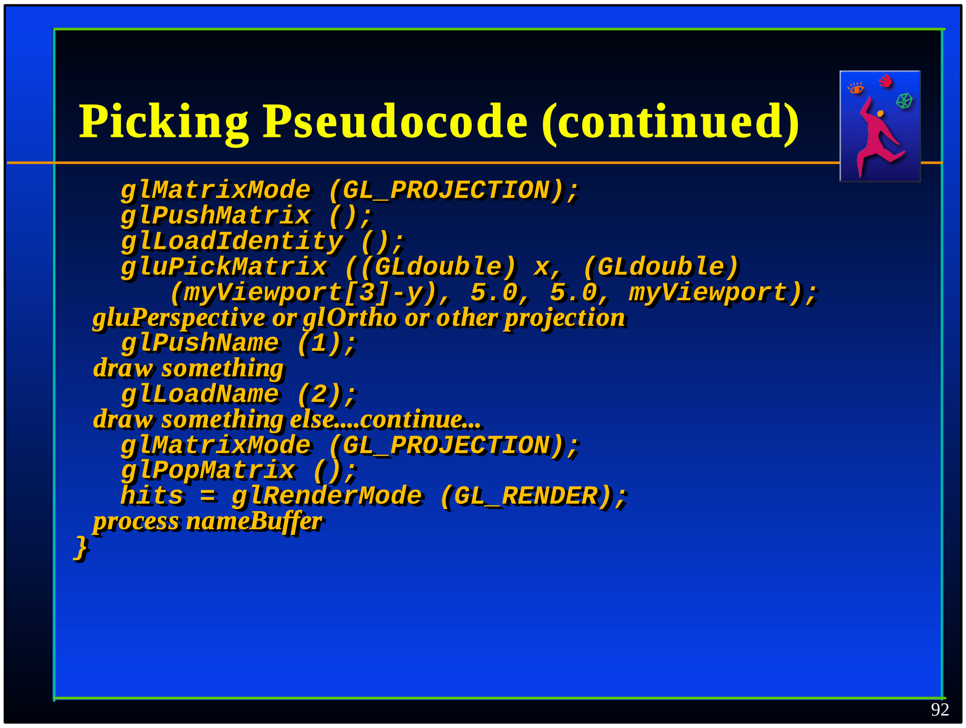
smallest and largest depth values

all the names

Picking Pseudocode

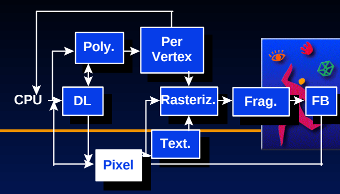
```
glutMouseFunc (pickMe);

void pickMe (int button, int state, int x, int y) {
   GLuint nameBuffer[256];
   GLint hits;
   GLint myViewport[4];
   if (button != GLUT_LEFT_BUTTON ||
       state != GLUT_DOWN) return;
   glGetIntegerv (GL_VIEWPORT, myViewport);
   glSelectBuffer (256, nameBuffer);
   (void) glRenderMode (GL_SELECT);
   glInitNames();
```



Be sure and push the first name you use onto the name stack. After the first name is pushed, it can be replaced by `glLoadName`.

Bitmaps and Images



Pixel-based primitives

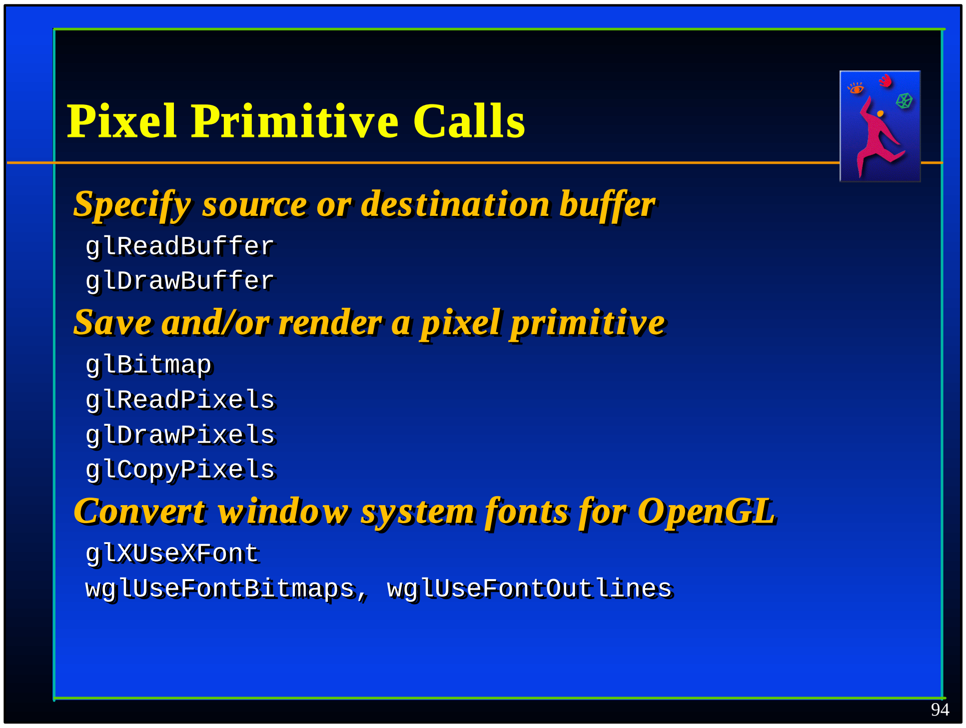
- bitmaps (one bit per pixel)
- pixmaps or pixel rectangles (many bits per pixel)
- texture images treated similarly

Use `glRasterPos*()` to position pixel primitive

- raster position transformed like geometry

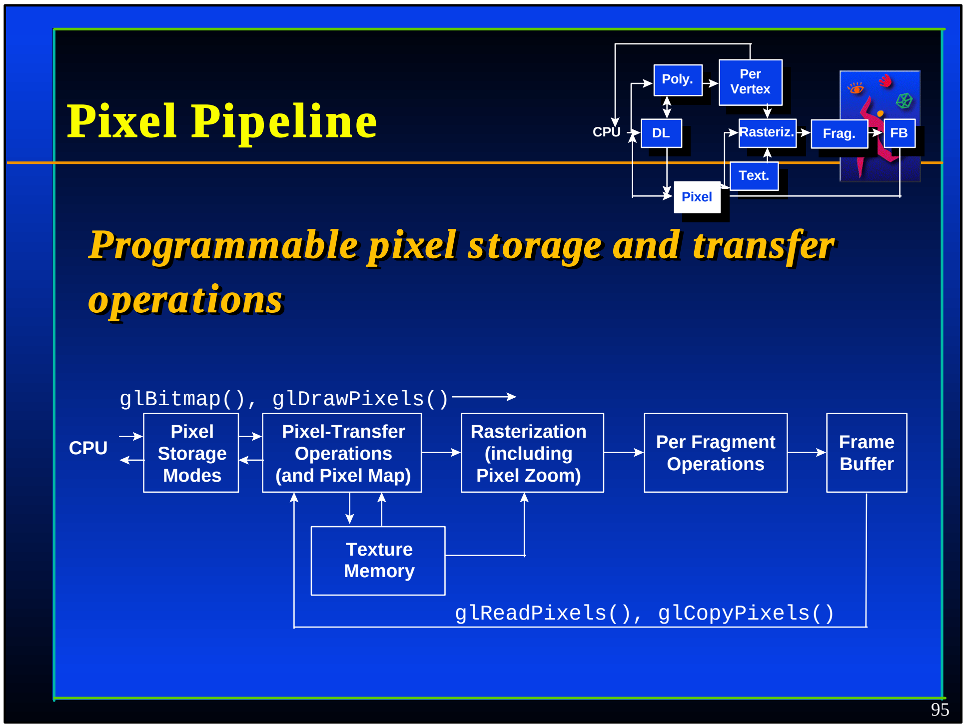
OpenGL does <u>not</u> encode/decode file formats (such as, GIF, JPEG, TIFF)

93



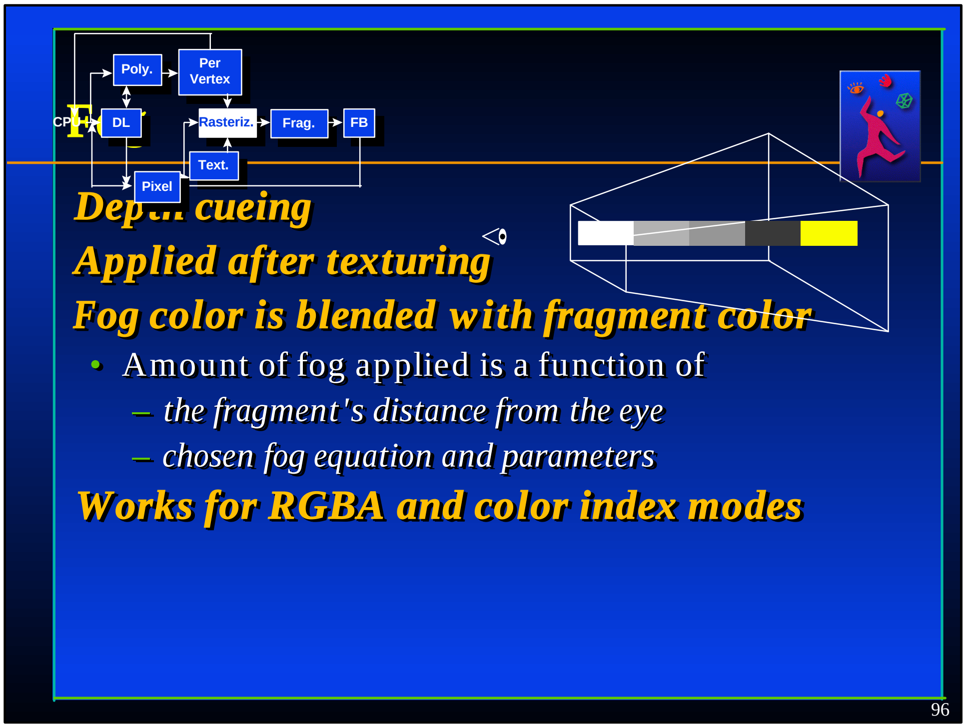
Source = incoming fragments

Destination = values (color, depth, etc.) which are already in the bitplanes



Warning: non-default values for pixel storage and transfer can be very slow.

For best performance, the internal representation of a pixel array should match the hardware. For example, for a 24 bit frame buffer, 8-8-8 RGB would probably be a good match, but 10-10-10 RGB could be bad.



Programming steps to perform fog

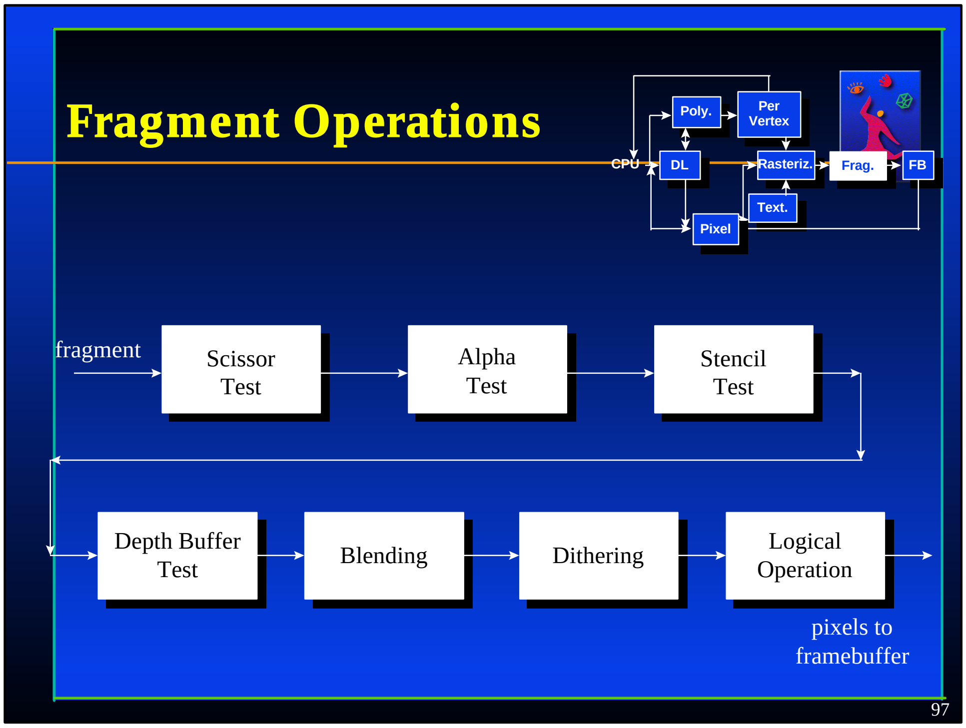
1) clear screen to fog color

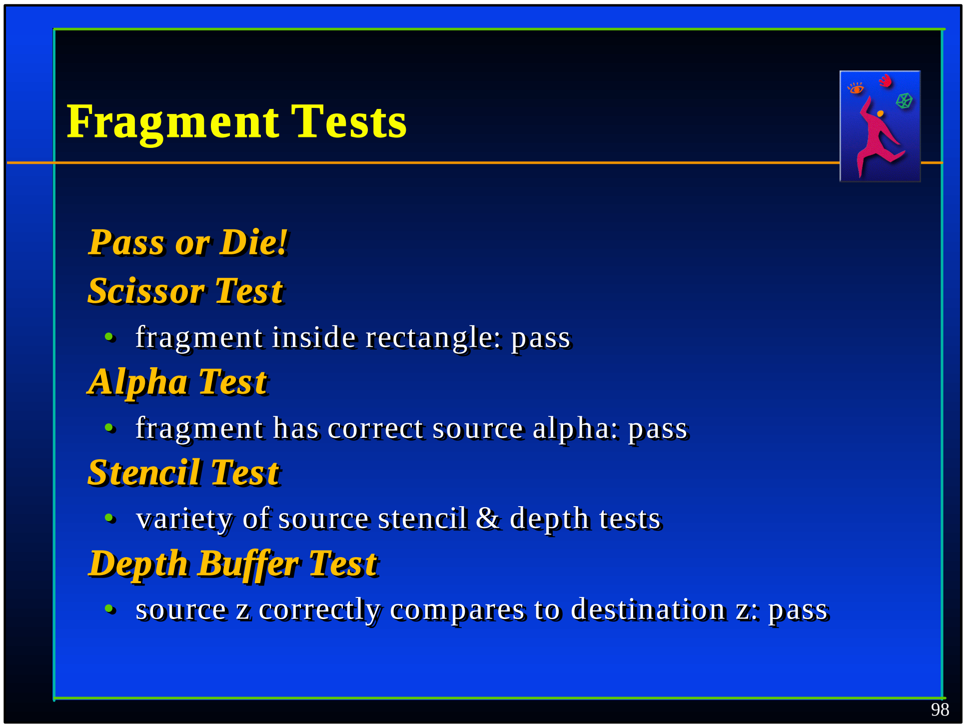
2) turn on fog with `glEnable(GL_FOG)`

3) use `glFog` to specify the fog equation to use (linear, exponential or exponential squared)

4) use `glFog` to specify parameters which affect fog density and color

5) use `glHint` to specify quality/performance tradeoffs


Fragment Operations
CPU
DL
Poly.
Per Vertex
Rasteriz.
Frag.
FB
Text.
Pixel
fragment
Scissor Test
Alpha Test
Stencil Test
Depth Buffer Test
Blending
Dithering
Logical Operation
pixels to framebuffer
97



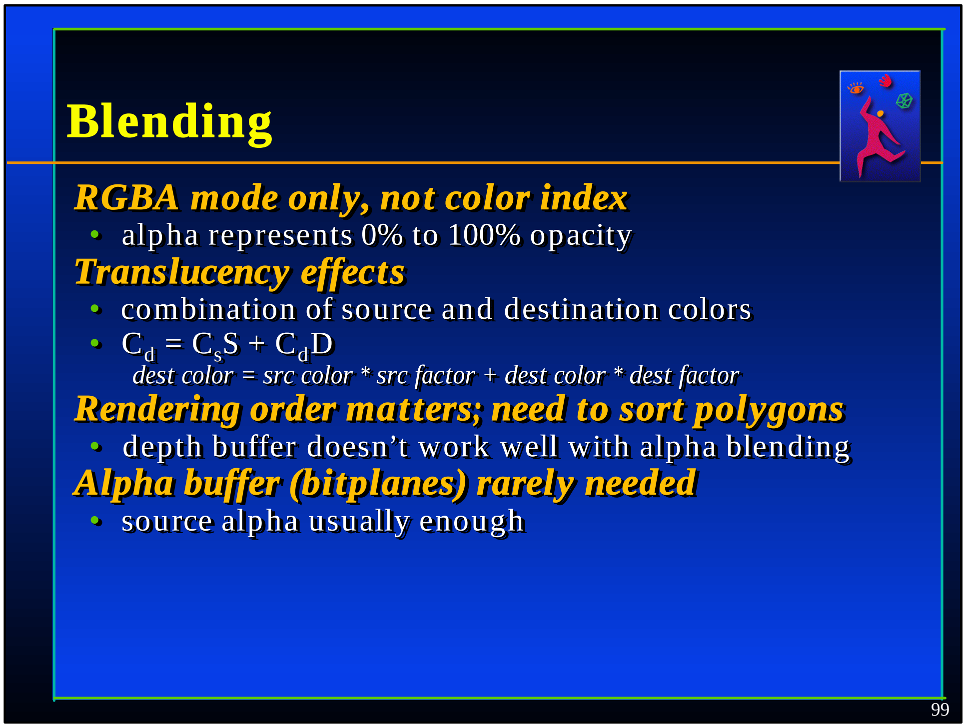
OpenGL routines which control these fragment operations:

```
glEnable (GL_SCISSOR_TEST)
glScissor (GLint x, GLint y, GLsizei width, GLsizei height)

glEnable (GL_ALPHA_TEST)
glAlphaFunc (GLenum func, GLclampf ref)

glEnable (GL_STENCIL_TEST)
glStencilFunc (GLenum func, GLint ref, GLuint mask)
glStencilOp (GLenum fail, GLenum zfail, GLenum zpass)
glStencilMask (GLuint mask)

glEnable (GL_DEPTH_TEST)
glDepthFunc (GLenum func)
glDepthRange (GLclampd near, GLclampd far)
glDepthMask (GLboolean flag)
```



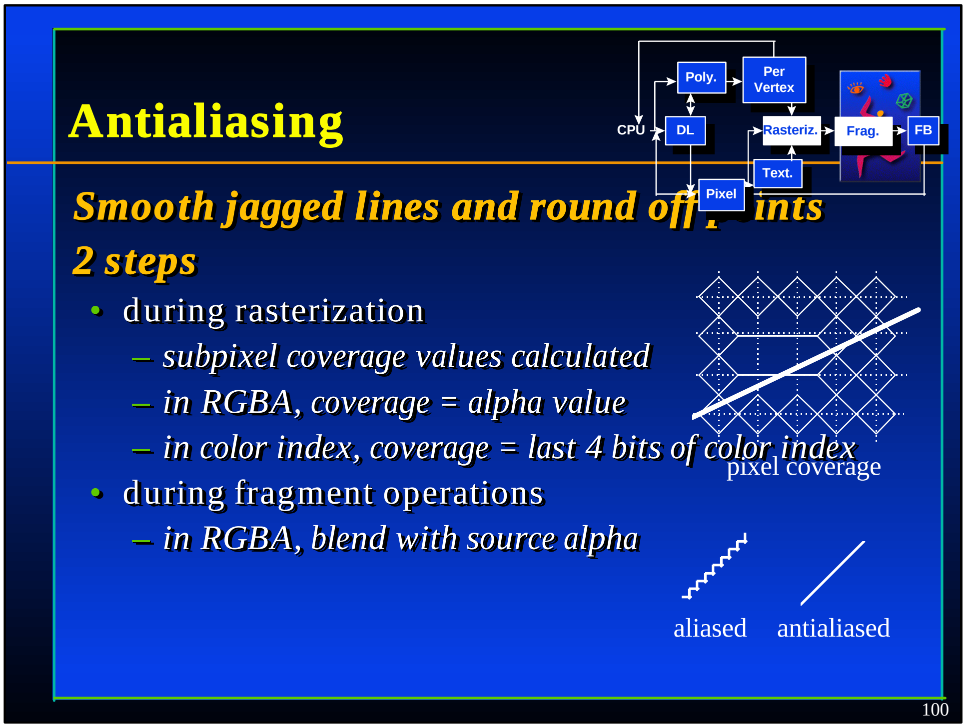
Blending routines:

```
glEnable(GL_BLEND)
glBlendFunc(sfactor, dfactor)
```

Some typical choices for sfactor and dfactor are:

```
glBlendFunc(GL_ONE, GL_ZERO)  /* default--no blending */
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
glBlendFunc(GL_SRC_ALPHA, GL_ONE)
```

There are very few uses for having actual bitplanes (destination alpha) to save alpha values. One use for destination alpha is to save a chromakey value for manipulating video images.



Routines for antialiasing lines:

```
glEnable (GL_LINE_SMOOTH)
glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE)
```

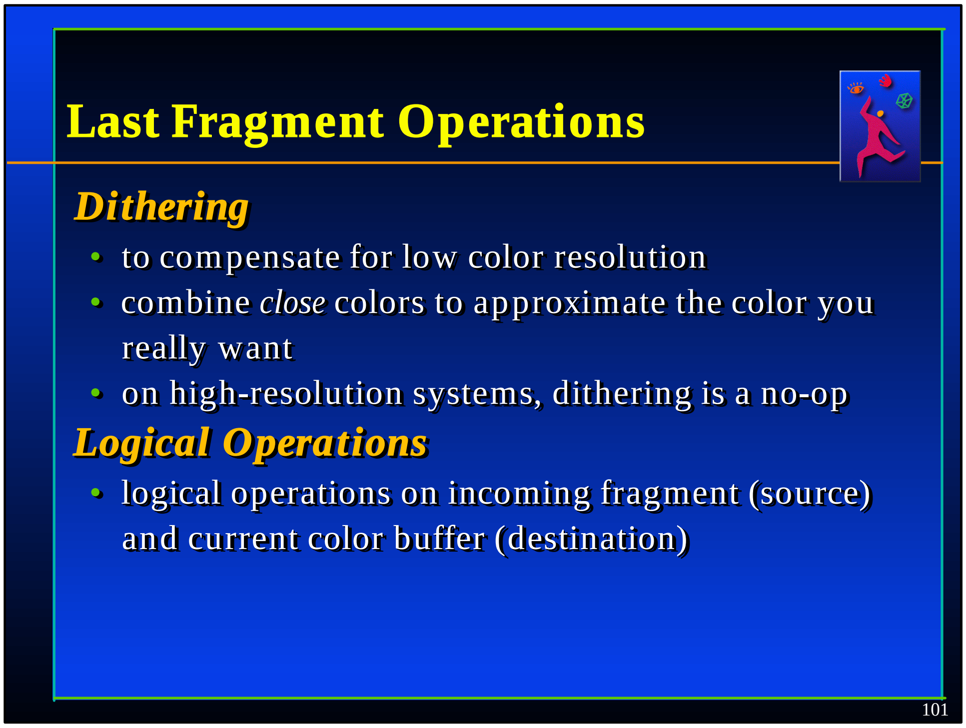
Routines for antialiasing points:

```
glEnable (GL_POINT_SMOOTH)
glHint (GL_POINT_SMOOTH_HINT, GL_DONT_CARE)
```

Routines for RGBA mode, only:

```
glEnable (GL_BLEND)
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

There is also `glEnable(GL_POLYGON_SMOOTH)` for antialiasing the edges of filled polygons, but it only works in RGBA mode. Also hidden surface removal with alpha blending raises additional issues. You might use the accumulation buffer as an alternate way to achieve full scene antialiasing.

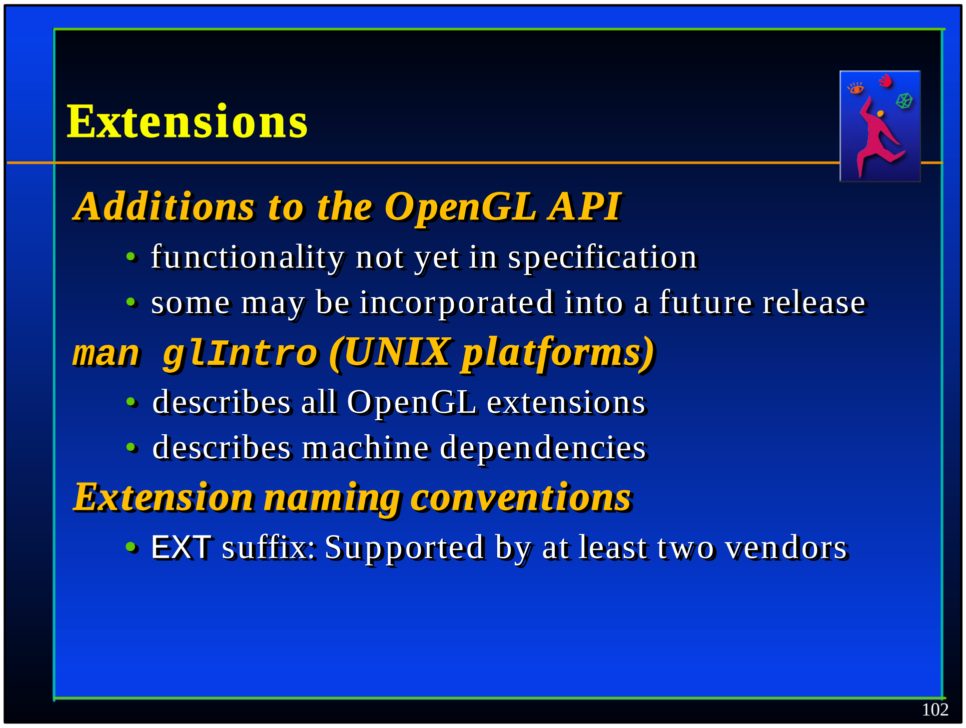


OpenGL routines:

```
glEnable (GL_DITHER)
```

`glEnable (GL_INDEX_LOGIC_OP)` or `glEnable (GL_COLOR_LOGIC_OP)`

```
glLogicOp (GLenum opcode)
```



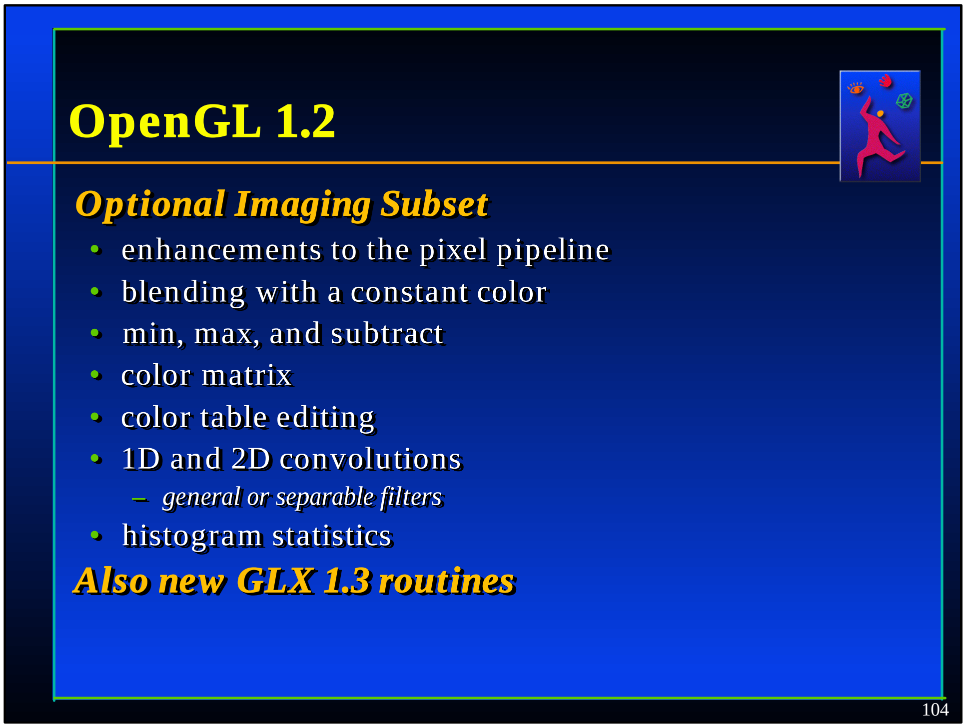
The availability of an extension can be queried by using `glGetString(GL_EXTENSIONS)` and looking for a string which is specified by the vendor. An alternate way of identifying the availability of an extension is to look for a constant at compile time (`#ifdef XXX`).

OpenGL 1.2

Mandatory New Core Capabilities

- vertex normals rescaled
- 3D textures
- texture coordinate edge clamping
- level of detail for mipmap textures
- specular highlights after texturing
- BGRA and packed pixel formats
- vertex array subrange operations

103



The OpenGL 1.2 imaging subset initiates a new concept for OpenGL functionality. The imaging subset is not mandatory for all OpenGL 1.2 implementations. However, if a vendor chooses to support it, they must support all functionality in the imaging subset.

Support for the imaging subset can be detected by using `glGetString(GL_EXTENSIONS)` and looking for the string `ARB_imaging`. However, it isn't an extension, and the functions and enumerants do not contain EXT suffixes.

The GLX 1.3 features include pixel buffers, more flexible frame buffer configuration, and support for read-only drawables (preparing for support for video).

Final Review: Typical Steps

open a window with specific visual/pixel format

- establish depth buffer and double buffer

to initialize

- read images from disk, load color maps
- tessellate polygons, load vertex arrays
- create display lists, texture objects, light sources

in resize()

- re-establish viewport, projection, and viewing transformations
- careful with the aspect ratio

105

Final Review (2)

in display()

- clear screen to background color
- initially push modelview matrix (usually)
- change states and render geometry & images
- finally restore modelview matrix (usually)
- swap buffers
- check for errors

optional routines for input devices or idle function

- input device may initiate picking

enter event processing infinite loop

On-Line Resources

http://www.opengl.org
- start here; up to date specification

news:comp.graphics.api.opengl

http://reality.sgi.com/opengl/opengl-links.html

http://www.microsoft.com/hwdev/devdes/opengl.htm

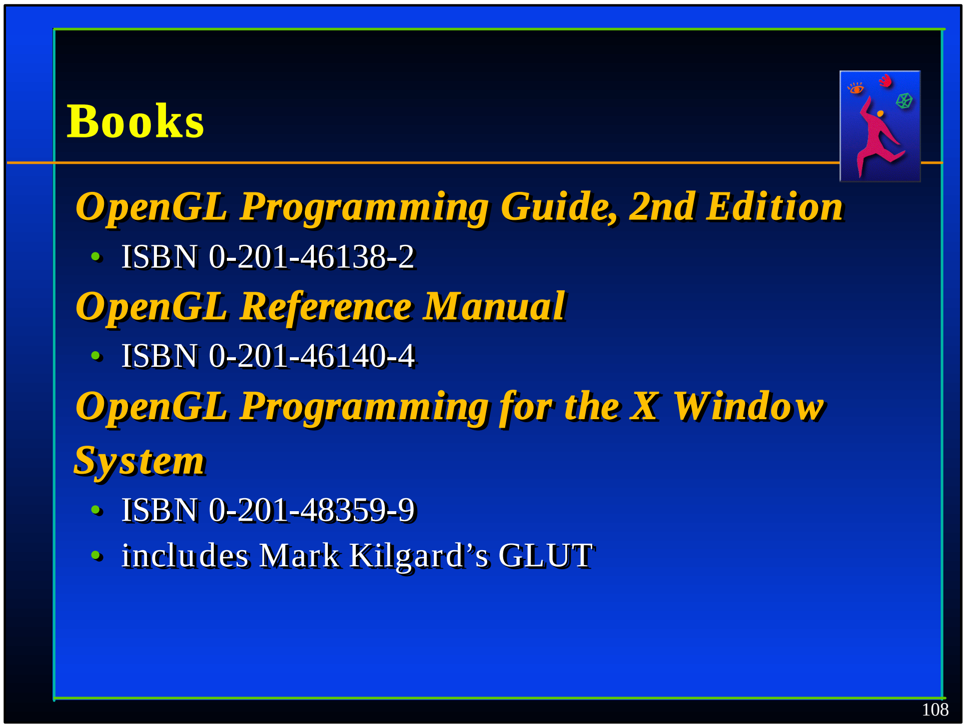
ftp://sgigate.sgi.com/pub/opengl

http://www.ssec.wisc.edu/~brianp/Mesa.html
- Brian Paul’s Mesa 3D

http://www.cs.utah.edu/~narobins/opengl.html
- very special thanks to Nate Robins for the OpenGL Tutors
- source code for tutors available here!

http://www.specbench.org/gpc/opc.static
- benchmarks

107



Other Books

OpenGL Programming for Windows 95 and Windows NT
by Ron Fosner, ISBN 0-201-40709-4

OpenGL SuperBible
by Richard S. Wright, Jr. and Michael Sweet, ISBN 1-57169-073-5

Interactive Computer Graphics: A Top-Down Approach with OpenGL
by Edward Angel, ISBN 0-201-85571-2

Thanks for Coming

Questions and Answers

mason@woo.com
shreiner@sgi.com

109